UiO : Department of Informatics
University of Oslo

A 3D stateroom editor

Andreas Alexander Gansen

Master's Thesis Autumn 2016

[simula . research laboratory]

A 3D stateroom editor

Andreas Alexander Gansen

Abstract

Today's software systems reach easily hundreds of thousands of lines of code, and such systems do frequently benefit from the use of state machines, which help in managing system complexity by guaranteeing completeness and consistency. To visualize such state machines, statecharts have been introduced, which also offer a formalism for orthogonal and hierarchical states. Many developers use state machines, but even with statecharts as a tool, it is a major challenge to keep an overview of the machine and its effects. Gaining an overview as a newcomer to a project is an even larger challenge. In this paper, we argue that a 3D statechart can alleviate these challenges somewhat, and present an editor for state machines that are given in SCXML, an XML notation for statecharts. This editor allows the user to explore the state machine by navigating freely in a 3D environment and make changes to the machine. The editor is a prototype and incomplete. It is an attempt to reflect the idea of having statecharts presented in 3D space.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Motivation

In our daily lives we encounter a huge amount of computer systems, knowingly and unknowingly. Since the beginning of their existence, they are getting smaller, faster, less power consuming and more portable. With the increase of computational power and networking, the expectations on software programs practically explode.

To meet the challenges of software's complexity, which is inevitable when adding features, developers introduced several methods to design software. Some of them, like design patterns or language features, help to organize software on a low-level to mid-level basis, but lack the ability to model high level logic.

As high level logic we consider the program flow ramified into sections where conditions determine the branches taken during execution. Such conditions can either be an expression, which evaluates the content of variables, or a waiting point, holding the current program until an event arrives.

Assuming a program has a modified file open and is waiting for user input. In that case, the flow control, viewed from the high-level logic, is halted until the user triggers an event. Presses the user the quit-program button, common software programs continue until they come to a branch with two possible directions, one where the program exits and another one, where a window rises, informing the user about the unsaved file. In the presented case, the content of the variable representing whether a file is modified or not would be true and the branch where the pop-up appears executed. This simple example shows two branches where the flow is directed once depending on an event and once depending on a conditional variable.

One may imagine the number of branching points and its resulting complexity of the Linux kernel; a software construct with currently 17 million lines of code[Linc]. Developers of such large software systems are often burdened with its complexity. Newcomers to such systems can be overwhelmed or even demotivated, especially if documentation is poor and supervision sparse.

State machines can help structure the code and make the program flow more comprehensible. In state machine terminology, branching points are named *state*s

while the flow between two states is denominated *transition*. For learning and editing, a graphical representation can be helpful, and Harel [Har87] formalized one known as *statecharts*. One textual notation for these statecharts is the XML notation, SCXML [Gro14], which has been used in some graphical editors [Linb; Line; Lind] and for which the interpreter *uscxml* [Rad15] exists.

Although visual editors exist, we find that they do not support developers well in understanding the state machine. The developers should be supported in making the causal connection between source code, system behaviour and a state machine's state.

Our goal is to provide developers with better support for understanding and memorizing state machines. The memorizing process shall not require a strenuous activity but rather a subconscious effect of interaction with system through the development tools. We expect that the better understanding of the system is achieved by letting the developer move freely through a state machine laid out in three dimensions. As a result, newcomers become acquainted with the system more quickly, while developers who are familiar with the system can resume their work faster after a break. Furthermore, communication between developers becomes easier because they can communicate with each other in metaphors when they can associate common concepts with the 3D layout of the state machine. For example, if a conglomerate of states is laid out like a tree, a developer may call another one's attention by saying: "Look behind the tree's trunk". Being able to name structures to facilitate communication is a valuable aspect from the agile software development area [Gam+95].

1.1 Mnemonic strategy

The human brain can store and recall information very efficiently if a mnemonic technique is used. One popular technique is *The Method of Loci*[Roe80] (MoL), aka memory palace. The idea is, that the person wanders in his mind along a well-known path, like the daily way to work or one's home. While moving along this way, the person can place images of items in salient location for later memorization. These images should be as vivid as possible. To recall the list of items later, the subject starts the imaginary journey again and picks up items as he progresses. If an item is not recalled, the journey, and therefore the retrieval process, is not disrupted.

In case of a three-dimensional statechart, which we call from here on *stateroom*, the path to wander is modeled by the stateroom's author. Software development is an iterative process. Hence, the author wanders through the room over and over again and *learns* subconsciously the stateroom by heart. Transitions are remembered as forks on the virtual *hiking trail*. States are analogous to locations in MoL, while imaginery pictures are considered as the result of actions occuring in a transition. This action could be visible on the screen; for example, a widget is focused, an item is selected or a text is changed. If the application controls a physical device, and the stateroom controls its

electrical motors and LEDs, the effect will even be visible outside of the computer.

MoL is used in general by constructing environments in front of one's inner eye, based on real-world experiences. A recent study [Leg+12] showed that MoL is similarly effective if the test subject uses a virtual environment instead of a well-known real world environment. This study supports our theory.

1.2 Problem Definition / Statement

This work is an attempt to develop an editor, which reads an SCXML file and renders a 3-dimensional representation of its logci on a 2-dimensional screen. We have to investigate how hierarchical states can be represented in 3D space. This is particularly interesting because, to the best of our knowledge, no reasearch has been publiched in that area so far. Since our goal is that a developer can experience a state machine's logic as much as possible, a way where the state machine is interpreted and becomes visible must be found.

1.3 Limitations

The editor is not in a state that we could conduct user-studies.

We were not able to present a useful visualization of orthogonal states. Compound states can be drawn, but when several ones are nested, there usability is questionable.

Transitions with multiple targets are ignored.

1.4 Main Contributions

We built the first editor for hierarchical state machines.

The presented work gives ideas on how states can be drawn in 3D, but also shows some dead ends.

The editor can connect via unix sockets to a server, to control simulated hardware in Blender.

This work was partialy presented under the topic *Storing coordinates of 3-dimensional staterooms* at the **scxml2016**-workshop in Brussel[Lina].

1.5 Outline

Chapter 2 Introduction to SCXML gives an overview of the classical automata and the way they are expanded by SCXML and statecharts.

Chapter 3 Mathematical background introduces basic mathematical concepts needed to calculate and present 3-dimensional objects on a computer screen. In particular is discussed how 3D objects are created and projected; Different kinds of billboarding are perceived by an observer;

Chapter 4 Introduction to Blender presents briefly the 3D animation tool Blender.

Chapter 5 Blender Game Engine gives an overview of Blender's game engine.

Chapter 6 Trial and error: Stateroom editor as game documents our attempt to build a stateroom editor with Blender's game engine.

Chapter 7 Introduction to OpenGL introduces OpenGL.

Chapter 8 Rendering pipeline describes the OpenGL pipeline.

Chapter 9 Design shows the widget to implement in the editor.

Chapter 10 Implementation in C++ contains details about the implementation of the editor in C++/OpenGL.

Chapter 11 Simulator explains the connection of the editor with a Blender-game.

1.6 Acknowledgement

I would like to thank my advisors Carsten Griwodz and Pål Halvorsen for guidance and feedback. Thanks to many people at the freenode irc channels, in particular to Olsen in #blender and derhass in #opengl.

A special thank goes to Carsten Griwodz, who gave me not just the opportunity to work on my own vision, he also provided constant and valuable support.

Part II

Theoretical Foundation

Chapter 2

Introduction to SCXML

In this chapter we present SCXML, a markup language for describing statecharts; a visual formalism for state machines introduced by Harel [Har87] in the mid 1980s. We choose SCXML as description language because Qt, the framework we introduce in chapter 10, offers a state machine based on SCXML.[1] Furthermore, SCXML is extensible and allows the saving of additional data besides the statecharts. We propose an extension for our purposes in subsection 2.5.5 and talk about the implementation in chapter 10.5.1.

In the next section we take a look at a simple state machine, while the subsequent section 2.2 gives a recap of classical state machines before we start with the presentation of SCXML in section 2.3; followed by the discussion of the SCXML notation in section 2.4.

The major part of this chapter is gathered from the official webpage of SCXML [Gro14], while information about automata in general is retrieved from the book [HMU01].

2.1 Basic Primitives

Many systems, like molecules or binary code, are constructed out of single, atomic entities, often called primitives. Such primitives may then evolve by adding attributes to more complex structures. In a very simple state machine domain we define three primitives: States, Transitions and Events. Such primitives can be assembled into a state diagram, illustrated in figure 2.1.

States are discrete points in a state chart and may be seen as branching or flow control units. In the introductory example a program was in a *waiting-state*, waiting for user input and afterwards, when the user pressed *Quit*, it reached an *is-file-to-store-state* where it waits until the user decides if the file is to store or to discard. A developer may introduce a new state when a decision, concerning the execution path that a program

[1] We assess in Qt's chapter that the latest version is not SCXML comform anymore.

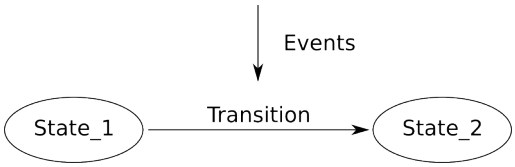


Figure 2.1: Basic primitives of a state machine.

should follow, is to be made. This is true for high-level considerations and not for if-clauses at programming language level.

Events are usually fired from an external source. They trigger the search on the currently active state for a matching transition. A transition matches when it is configured to react to the incoming event.

Transitions build the pathways between the states through the state machine. We say a transition is enabled, taken or traversed. When a transition is traversed, its source state becomes inactive while its destination state becomes the active state. Usually a transition is enabled when a corresponding event occurs.

2.2 Automaton

States, events and transitions are the core elements of *automata* or *finite state machine* (FSM). We will not engage in a complete discussion about FSMs, but rather sensitise ourself by looking at their characteristics to emphasize the properties of SCXML in the subsequent section.

FSMs are categorized into deterministic and non-deterministic state machines. Both consists in general of the following entities:

- Finite set of states
- Finite set of input symbols (events)
- Transition function
- Start state
- Set of final states

A system, interpreting a state machine, has usually an entity whose content holds the present state of the state machine. To facilitate the discussion we introduce the variable *current_state* for that purpose. The *finite set of states* contains all states the FSM operates in, hence, the current_state contains always one element of that set. The *start state* marks the beginning of the interpretation of a state machine. It is the first entry current_state holds. States marking an end of an automaton are gathered in the *set of final states*. However, FSMs allow transitions leaving a final state. *Input symbols* is the alphabet the state machine expects to react to. Together with a state they form a (symbol,state)-tuple as argument for the *transition function*. This function determines

if the FSM is a deterministic (DFA) or non-deterministic (NFA) automaton. For a DFA, only one state is returned, because a deterministic automaton can only reside in one state at a time. The transition function for NFAs return a set of states, empowering the state machine in residing in multiple states at the same time. Every NFA can be transfered into an equivalent DFA. The power of the NFA is to express complex systems with less states.

Moore[2] and **Mealy** extended independently the FSM and added a notation for output. In figure 2.2, both notations are shown. The Mealy machine generates output when a transition is take, while a Moore machine generates output when a state is reached.

We do not discuss these machines further, but during the discussion of SCXML, we take the opportunity to emphasize the concepts by comparing SCXML to these classical machines.

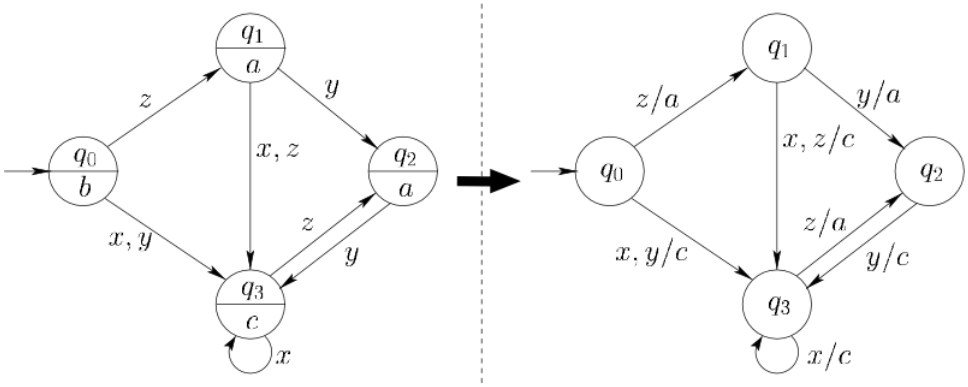


Figure 2.2: A Moore automaton(l) and an equivalen Mealy automaton(r).[May14]
with q_n: states; x, y, z: input; a, b, c: output

[2]Named after the mathematicians Edward F. Moore and George H. Mealy.

2.3 Expanding to SCXML

State diagrams are the visualization of FSMs. Harel enriches [Har87; Har88; HN96] state diagrams with **orthogonality**, **depth** and **broadcast-communication** to a visual formalism named **Hierarchical State Machine** (HSM) or **Statechart**. These features allow a corresponding state machine to have several states active at the same time, nested states, which introduce a parent-child relation between states, and a communication factility to raise events internally or in remote state machines. SCXML offers an XML notation for HSMs as well as a set of algorithms to interpret them. Our focus lies on that part of SCXML, which describes the notation for HSMs without broadcast-communication. We do not discuss the algorithms for the interpreter, but we look into those parts necessary to understand the semantics of statecharts.

Before we look at the notation tag by tag, we introduce some key concepts of SCXML to get an overview of the standard.

SCXML's states can be partitioned into three classes:

- **Atomic** states are discrete states and are equivalent to FSM-states.
- **Composite** states, are superstates consolidating child- or substates. A composite state is either a parallel state, where its children become active together with the paralle state. Or a composite state is of type compound state, forming a hierarchy between states with an atomic state at its bottom. Children of a parallel state form an and-relation, because if one thread is active, all are; while the atomic states in a hierarchy have an exclusive-or-relation, meaning only one atomic state can be active at a time.
- **Pseudo states** never become active but they redirect a transition, when activate, to the set of states which become active instead.

In SCXML, a state is denominated *active* if the current set of states contains that state. All active states together are collectively denominated *configuration*. We refine this term along this chapter and give a final definition in section 2.5.6.

In a HSM, a state can have an arbitrary number of siblings, descendants and ancestors. Therefore, it can be discussed like a family tree. Figure 2.3 illustrates 4 states that have among others the following relations: **A** is parent of **B** and **B** is child of **A**; **A** is ancestor of **B**, **C** and **D**; **B** is sibling of **D**; **C** is descendant of **A** and **B**. States **A** and **B** are **compound state**s while states **C** and **D** are **atomic state**s. An active compound state has at least one child active, ergo a branch ends always in an atomic state. When a child state is active, then its lineage is active too and all its ancestors are part of the configuration. That includes also the root state of SCXML, which is the tag <scxml>. FSM's initial states are qualified states where the state machine resides in after startup. So, SCXML allows to have a transition upon startup, entering the *initial* state.

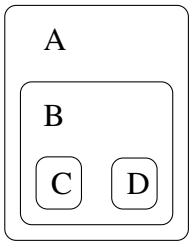
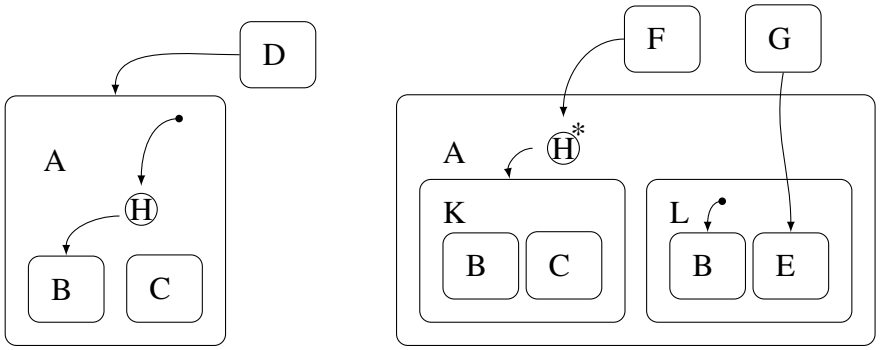


Figure 2.3: Nested compound state.



(a) Type shallow with default state **B**. **A**'s default state is the history state, history is applied every time **A** is targeted.

(b) The deep history stores either K(B|C) or L(B|D) and is applied from **F**'s transition. **G** ignores the history and activates state **E**.

Figure 2.4: History states of type shallow and deep.

A *final* state is a state without transitions.When it is reached, an automatic event is cast internally. This is completely different from FSMs where states can be marked as an *accept* state. But these accept states only indicate that the machine accepts the order of the received events.[3] And because they are allowed to have transitions, the acceptence status may be revoked when a matching event occurs. A final state, however, has no transitions. But it still can be exited through a transition of an ancestor, unless it is a direct child of the *scxml*-root wrapper, the document's root state.

Pseudo states, namely *initial* and *history*, redirect the transition flow of the state machine and do never occur in a configuration. An initial state redirects the flow to a child state when a composite state P becomes active, and the activation is in such a way that P is targeted and not one of its children. For example, in figure 2.4b **L**'s initial state is **B**. Because **G** targets **B**, **L**'s initial state is ignored if **G**'s transition is taken. Initialization is specific for hierarichal states and FSMs do not have such a setting. History states store a configuration when their parent is exited and restore the configuration when it is activated. Note that a history state does not restore the previous state upon its parent state activation. It must be targeted, either as parent's default state or by another state. A history state can be configured as *shallow* or *deep*. Setting it to *shallow* (see figure 2.4a) stores only the parents current active children. Is the type is set to *deep* (shown in figure 2.4b) the complete configuration to the final atomic state is recorded. FSM's do not have any notion of history.

A parallel state is divided into orthogonal *regions*. Each region is activated when the parallel state becomes active. This empowers SCXML to have multiple atomic states in a *configuration*; making SCXML equivalent to a NDA. Figure 2.5 shows the conjunction **Y** of the two compound states **K** and **J**. If **Y** becomes active, the initial transitions to **A** and **G** are taken. Each new incoming event is processed from each region independently: Suppose we have the configuration **scxml, Y, K, J, B, E** and α is triggered, we end up

[3] In FSM terminology, events are named symbols and a chain of symbols is named string.

in the configuration **scxml, Y, K, J, A, F**. A parallel state may be exited by making a transition to a state that lies outside of the region **Y**. This is shown in the figure with transition *f* to state **X**. A *final* state can be used to stop further executing a region (not shown).

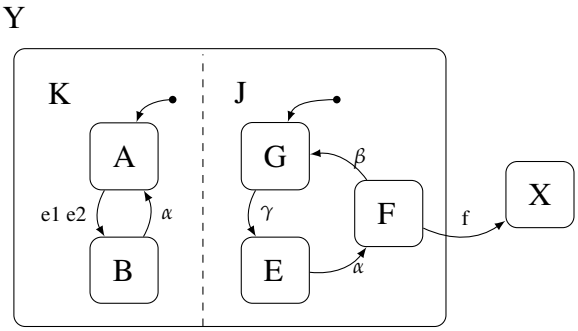


Figure 2.5: Parallel state with two orthogonal regions (*K*, *J*).

To react on a specific stage in state machine traversal, Mealy and Moore machines can generate output either when a transition is taken (Mealy) or when a state is reached (Moore). SCXML combines and extends this feature under the topci *executable content*. Executable content may be attached to a transition or as handler to a state. Two handler types exists for states, onentry and onexit, which are called when a state is entered or exited, respectively. This is comparable with constructors and destructors in OOP object. Transitions' executable content is performed during state change, between the execution of the onexit and onentry handler. But executable content does not just generate output, it allows to execute code snippets, which is elaborated further in section 2.5.3. Figure 2.6 shows three states and two transitions. Assume state **A** has an exit handler **exit-H-A** and state **B** an entry handler **entry-H-B** and the transition **e1** has executable content **e1-C** attached; taking transition **e1** would result in the execution of **exit-H-A, e1-C, entry-H-B** in that specific order. If under the same assumption **e2** is traversed, the execution of **exit-H-A, entry-H-B** would be performed. The one to the left is enabled by events *e1* and *e2*, the one to the right by event *e3* only. Transitions form the unidirectional link between states. A transition is triggered by an *event*. It may react to multiple events and target multiple destinations.

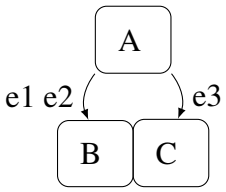


Figure 2.6: Two transitions targeting **B** and **C**

Considering plain states in FSMs leadss to the cognition how simple they are: Only one of them can be active. In HSMs, however, where states can be nested and several states may be active simultaneously, it is worth, if not even necessary, to discuss states and there interrelations without transitions, events or other features. For example, a nested state scenario is considered: An atomic state at the bottom of the hierarchy is active and a transition to a target on a different branch is to be taken. Several ancestors are here affected. To calculate which of their entry/exit handlers must be executed, in what order, during state switch, it is often necessary to determine the **Least Common Compound Ancestor (LCCA)**. The LCCA is determined by finding the target's closest compound state that is also contained in the *configuration*.

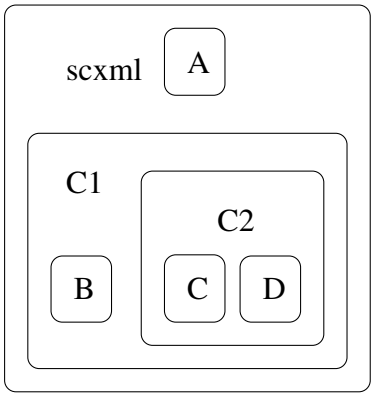


Figure 2.7: Nested compound states with the root wrapper SCXML.

For state **D** in figure 2.7 we climb up the hierarchy until we find the next compound parent, which is **C2**. In a transition from the *source* state **D** to the *target* state **B**, we identify the configuration: $D, C2, C1, scxml$[4]. **B**'s closest compound state is **C1** which is contained in the configuration. So, **C1** is LCCA for this transition. The outermost and omnipresent LCCA is per definition the *scxml* root element, explained in chapter 2.4.6. Since the LCCA is the closest parent of the states involved in transitions, we can conclude that the LCCA itself is never left and re-entered during transitions. The LCCA becomes more important when we talk about transition *types* later in this chapter.

[4]The reader is encouraged to return here after finishing the document and explain, why A and B can't be active, when D is.

2.4 SCXML notation

This section lists and explains in brief the SCXML notation. We start each section by introducing the **attributes** followed by the **child** elements.

Before we start, we introduce the common attribute **id**[5] because it is used frequently throughout the standard. It holds a unique identifier, which is used when the parent element is targeted by a transition.

2.4.1 State

Atomic states and compound states are incorporated in <state>. Which one a <state> represents depends wether it has a as child state or not. A parallel state is almost identical to <state>, but has its own tag, <parallel>. In section 2.4.2, we describe the differences.

Attribute-tags

A <state> can be configured with two optional attributes. Besides the **id** attribute, it has an **initial** attribute, declaring an initial configuration. For example, transition to **C** in figure 2.8 shows the graphical representation of the start-tag: $< state\ id = "B"\ initial = "C" >$. This option preclude executable content in the body of the transition. To overcome this shortcoming an *initial* child-tag is introduced in the next paragraph. In SCXML, if no initial state is given, then the default initial state is the first state in document order.

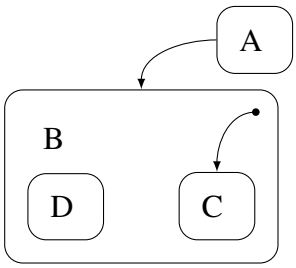


Figure 2.8: Compound state **B** contains an initial tag with a target to **C**.

Child-tags

In brackets is the range specified how often an element may occur.

- <onentry>/<onexit>[0-*] These two executable tags are processed during state entry and exit. They may occur several times and are worked up in document order. If a previous handler aborts on error the subsequent ones are not affected. Executable content is discussed in section 2.5.3.
- <transition>[0-*] A state may have an arbitrary number of transitions to other states. Transition are discussed in 2.4.5.

[5]Possible characters for an id are defined at XML1.0 under *[5] Name*. In general it is save to use: (Letter | '_' | ':')*

- <initial>[0-1] Chapter 2.3 defines that a legal configuration branch must end in an atomic state. We also clarified that the *initial* attribute denotes a child state (or states) to be entered when the compound state is targeted itself. This tag needs a transition as child and allows thereby to execute content as default initialization.
- <state>[0-*] The list of child states. If present, this becomes a compound state.
- <parallel>[0-*] Introduces a parallel substate and promotes the state to a compound state. Chapter 2.3 discusses parallel states.
- <final>[0-*] List of final states.
- <history>[0-*] A <history> element, see section 2.4.4.
- <datamodel>[0-1] Can be used to set variables inside the scripting language. Datamodel is discussed in chapter 2.5.2.
- <invoke>[0-*] Is used for external services and covered in section 2.5.4.

2.4.2 Parallel

The parallel state uses the tag <parallel>. It is equivalent to <state> but misses the attribute **initial** and the children <initial> and <final>.

2.4.3 Final

The final state <final> provides a single **attribute**: *id*.

Its child tags are <onentry>, <onexit>, as discussed in section 2.4.1 and a new tag <donedata>. With <donedata> some data consisting of <content> and <param> can be returned to its callee.

2.4.4 Pseudo states

History History states use the tag <history> and two optional attributes: **id** and **type**. The type determines if the history to be stored is of type *deep* or *shallow*, as discussed in section 2.3.

As child is allowed an mandatory a single transition <transition>, which is discussed in 2.4.5.

Initial Initial states are inclosed in an <initial> element and has no attributes.

The only child holds the initial configuration and is is a <transition> element, discussed in 2.4.5. This transition is mandatory.

2.4.5 Transition

SCXML's transitions are triggered like FSM transitions, through an event. Furthermore, SCXML transitions can have several targets, equivalent to NDAs. These are necessary to address parallel states.

The following attributes are vastly beyond FSM's capabilities and lead to more complex behavior.

Attribute-tags

- 'event' Is a space separated list of *event descriptors*, which lead to a traversal of the transition. Event descriptors are further defined in section 2.4.7.
- 'cond' A conditional expression must be evaluable to *true* or *false*. Only if this, so called *guard*[Sam09, p. 65f], is *true* the transition might be selected. The expression is evaluated solely on event occurrence.
- 'target' Contains a list of destination state-identifiers that can be reached by taking this transition. Multiple, space separated entries may be defined to target different states with a single event.
- 'type' Transitions are of type *internal*[6] or *external* (default). The type is only interesting for *compound states* and regulates which entry and exit handlers are called. When the *external* switch is chosen, all exit-handlers between the current configuration up to the LCCA (excluded) and all entry-handlers from the LCCA to the next configuration are called. The *internal* parameter is slightly more complicated. It prevents executing the handlers of states, which would be left and re-entered again to reach the target state. Figure 2.9 shows a simple example. Suppose the current configuration is **C, A, scxml** and event *t1* is triggered. Since *t1* is not in any of **C**'s transition event descriptors, the event is "passed" up to **C**'s parent **A** for processing[7]. **A**'s only transition matches the current event and is therefore enabled. Because **B**, the target, is child of **A** like **C**, that makes **A** the *LCCA*. Depending on *t1*' type, we have two execution paths: *external*: C1.onexit, t1.body, B.onentry
internal: t1.body

A transition's only **child** element is executable content, see section 2.5.3.

[6] The UML specification uses the term *local* instead of *internal*.
[7] How transitions are selected is described in section 2.5.1

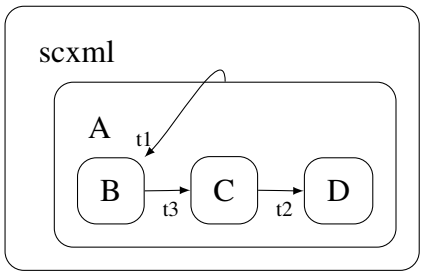


Figure 2.9: Is **C** active and *t1* enabled, *t1*'s *type* determines if **A**'s exit handler is called.

2.4.6 The root tag <scxml>

Each SCXML document provides the <scxml> tag as root tag. It is a slim compound state with the following attributes:

- initial - The optional initial tag identifies the state to set active on start-up.
- name - For informative use only a name.
- xmlns - Contains "http://www.w3.org/2005/07/scxml".
- version - Contains "1.0"
- datamodel - SCXML can use data structures and evaluable statements for conditional variables. Ignored in this document.
- binding - Defines at which point data elements are initialized. Beyond the scope of this document.

Children

The children <state>, <parallel>, <final>, <datamodel> are identical of those of a <state> tag. The additional <script> tag allows to run a script during load time to initialize the datamodel and is out of the scope of this document.

2.4.7 Event descriptors

Event descriptors are either generated by the state machine or received from an external entity. An event descriptor has the form: "t1.t2.t3" with sub-identifiers delimited by a period. A transition will match if a sub-string matches from the beginning. For the string above, "t1", "t1.t2" will match, "t2" won't match. Event descriptors are reflected as structure *_event* in an executable content environemnt and in condictional expression, like the attribute 'cond' in transitions. See chapter 2.5.3 for more information.

2.5 SCXML interpretation

A SCXML-file is interpreted by a *processor* and acts like a pure *event processor*. The detailed algorithms can be found in the standard at chapter D Algorithm for SCXML Interpretation and are not in the scope of this work.

Nevertheless, the next subsections provide an understanding on how transitions are taken, how we can benefit from executable content and how SCXML can extended. We use this knowledge to communicate with simulated hardware in chapter 11.

2.5.1 Transition selection process

In the previous sections, we have illustrated how complex structures with *states*, *parallel states* and *compound states* are expressed. These are connected by *transitions*. To traverse such structures in a deterministic way, the right transition according to the occurred event and the definition of the current *configuration* has to be taken. A transition is taken when,

- the transition's source state is active and
- no *cond*-guard is given OR the conditional variable is evaluated to true and
- no *event* is specified OR the event occurred
- no active descendant of the transition's source state fulfills the previous requirements.

While the first 3 points are easy to understand, the 4th point might raise some confusion. When an event occurs, the state-machine tries to find first a match in all active atomic states' transitions. If none of these atomic states provides a proper transition, the state machine climbs up the hierarchy and looks at its parent's transitions and their parent's transition and so forth. If no transition is found, the event is discarded. If, however, two or more active states don't match but their LCCA, then all states are exited and the transition is taken. In some constellations, more than one state may provide a proper transition. In such a conflict, which we designate *intersection*, the state sooner in document order wins.

Listing 2.1 gives an example where transitions' triggers intersect. Transition *s1* is taken in case event *e* is fired and the conditional guard $c1 == 1$. If the guard evaluates to *false*, the 2nd transition is enabled. The last transition is set as default event and is traversed on any event independent of the guard.

```
<state id="s0">
    <transition event="e" cond="c1==1" target="s1"/>
    <transition event="e" target="s2"/>
    <transition event="*" target="s3">
        Some executable content
    </transition>
</state>
```

Listing 2.1: Transitions with a guard, executable content and different targets but intersecting events.

2.5.2 Data Model

A data model extends the SCXML standard with an abstract way to access an underlying scripting system (e.g. ECMAScript). The processor is free to support an additional data model but must support the *null* data model, see section Null Data Model below. Appendix B of the SCXML standard lists 4 capabilities a conformant data model must provide:

- **Boolean expression language** is used in the 'cond' attribute of <transition>, <if> and <elseif>. There executable children will only be executed if 'cond' evaluates to true. The predicate **In()** must be provided in addition to the platform specific language.
- **Location expression language** pinpoint to a location in the data model. It can be referenced in the 'location' attribute of <assign>, <param>, <send> and <invoke> tags. Its notation is platform specific.
- **Value expression language** defines expressions in attributes of the type *value expression*, like 'eventexpr', 'targetexpr' or 'expr', in many tags (e.g. <send> <log> <param> <data> <assign> <content>). The evaluated result of an expression is then passed as the data item.
- **Scripting language** embeds a script which has full access to the scripting environment (global variables, functions, System variables (see section System Variables below, etc.).

System Variables

A platform specific data model must also provide the following system variables:

- **_event.** Holds the event currently being processed.
- **_sessionid.** An identifier for this session.
- **_name.** Is the 'name' attribute of the scxml-element.
- **_ioprocessors.** Lists all available I/O Processors, discussed in section 2.5.4.
- **_x.** Is the root element for platform specific system variables.

Null Data Model

The null data model does not support any functionality listed above nor the <foreach>-tag. But it implements the predicate **In(x)**. This predicate returns a Boolean depending on, if *x* is part of the current state configuration. This is interesting because it can be used inside a *cond* attribute, and check if a parallel region is inside a specific state.

Provided Tags

There are several tags in SCXML, which allows a state machine to communicate either internally, or with a remote system. We present only <param> and <content>, since these two are the ones used for communication in chapter 11.

The <param> tag can be used to send a key value pair, while the <content> tag allows to send plain text to the counterpart station.

2.5.3 Executable Content

Executable content can occur as children of the elements <onentry>, <onexit> and <transitions> and is processed when the element becomes active. The content being executed can provide functionality beyond the standard SCXML.

Section 4 of the SCXML standard discusses executable content. We only use the tags <invoke> and <send>, discussed next.

2.5.4 Communication

The processor can be extended to communicate with Event I/O Processor two tags: <param> <content> A state machine without any communication to the outside world is of little practical use. Therefore the SCXML standard offers in chapter 6, *External Communications* two capabilities to interact with the embedding processor. The processor execute an internal Event I/O Processors depending on what the SCXML-file being executed defines. These capabilities are platform-specific, but a few, to send events to other SCXML-sessions are built-in.

The **<send>**-tag is earmarked for sending messages to a specific destination. It can be defined wherever Executable Content is allowed, that is as child of <transition>, <onentry> and <onexit> tags. This emphasizes the fire and forget philosophy of <send>. Various attributes can be used to pass information to the I/O processor. We list here only a selection:

- **target** and **targetexpr** points to a URI to where the message should be send to. While targetexpr
- **type** this attribute addresses which Send-I/O Processor the system uses to deliver the message. A default value http://www.w3.org/TR/scxml/#SCXMLEventProcessor can be set, so that the message is send by a platform specific method.

As children, up to one <content> and an arbitrary number of <param> tags are allowed.

An **<invoke>**-tag starts a *service* and is only allowed as child of a <parallel> or <state> tag. It executs as long as the invoking state is active.

- **id** is used to identify a specific instance of the I/O Processor. Can be used in the <send> tag.
- **type** as for <send>, this attribute addresses which I/O Processor the system creates, but this time it is Invoker I/O Processor.

2.5.5 Extending SCXML

The standard explicitly allows in section *Interoperability considerations* to be extended in accordance with its MIME type `application/scxml+xml`. That means that SCXML compliant processors have to ignore any XML elements that differ from the SCXML standard.

We extend SCXML by the following attributes:

- The <state> tag is extended by the attributes: xpos, ypos, zpos. These attributes store the Cartesian coordinates of that element. Each attribute contains a hex string, representing a float value encoded in the widely used IEEE754 standard. This allows to retrieve the exact value, without losing precision.
- The <scxml> tag is extended by the attribute: stateroom. Its only option is *true* and it indicates that the document presented has positional information, as declared in the previous items.

For simplicity, we do not encapsulate these tags into a namespace, although that would probably be the better practice.

These extension are written by the SCXML serializer discussed in chapter 10.5.5 and read by the parser presented in chapter 10.5.1.

2.5.6 Configuration

A configuration contains all active states in a SCXML state machine.

- At least one atomic state is active.
- All ancestors of each active atomic state is contained in the configuration.
- If among the active state is a parallel state, than its children are part of the configuration.
- Each active compound state has one active child state.

2.6 Final Note

In this chapter we discussed the capabilities of SCXML to express complex state machines. We also peaked inside the interpretation of SCXML and how the interpreter can communicate with the outside world. But we did not dive to deep into the internal algorithms used by the interpreter. This is described in part D of the standard and not completely necessary for our case.

Chapter 3

Mathematical background

In the introductory part, we set our goal to develop a software application with the main purpose of organizing staterooms in a 3D environment and render them onto a 2D screen. Users shall be able to interact with these staterooms and move freely in the 3D world. This chapter presents the mathematical background for the projection of 3D models onto computer screens, how a camera can be rotated smoothly and how one ensures that a state, which is a 2D element, always faces an observer. Besides this, we look into two algorithms used to lay out graphs in 2D as well as in 3D.

3.1 Coordinate Systems

To understand how 3D models are projected on a 2D surface and how camera movement is simulated, we have to look at different coordinate systems and how vertices are transformed between them. This section builds the foundation for 3D processing and forms in particular the basis for some design decision made in chapter 9 and 10.

There are numerous sources available about this subject. Nevertheless, this chapter is based on the following literature: [Len04, ch. 3, 4], [AMHH08, ch. 2], [DP11, ch. 10.3]

Before we explore the different coordinate systems and how to convert coordinates between them, we recap the basics of linear algebra.

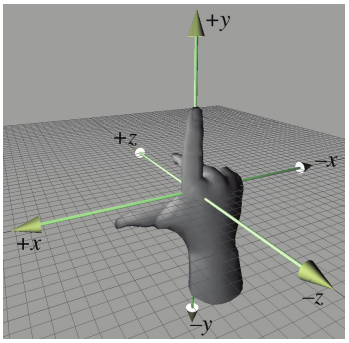


Figure 3.1: The 3 axis of the right-handed 3D coordinate system.[DP11, p.16]

In a 3D environment, several Euclidean $\mathbb{R}^3$ spaces with the Cartesian coordinate system are used for the transformation of 3-dimensional scenes into 2-dimensional images.[1] The axes of such a system (x, y and z) are mostly configured right-handed; see figure 3.1. It is called right-handed because if the X-axis is assigned to the thumb of a right human hand and the Y-axis is

[1] The transformation process involves also some spaces in $\mathbb{R}^2$, which are considered as prerequisite.

assigned to the index finger, then the middle finger represents the Z-axis if directed perpendicular to the palm. All fingers point to their corresponding positive direction.

Positional indicators in $\mathbb{R}^3$ are called **vertices** and have the form $P = (x,\ y,\ z)$. However, before vertices are transformed, they are extended to *homogeneous* coordinates by adding a 4th component w. A vertex is then defined as a quadruple $P = (x, y, z, w)$. This has several advantages. Coordinates in homogeneous space can represent 3D points at infinity ($P_{inf} = (x, y, z, w = 0)$), which can then be treated as any other point. In Euclidean space, it is necessary to check for this anomaly. Furthermore, 4-component vertices may be multiplied with a 4x4 matrix; applying scaling, rotation and translation at once. Non-homogeneous coordinates cannot combine translations with scaling and rotation in a single matrix. Hence, they miss the important optimization to chain matrices together as elaborated below.[Shr+13, p. 215ff] [Ago05, p. 134ff]

Matrices can be decomposed into their functional parts. Equations 3.1-3.3 show which part leads to what kind of transformation. There is the translation matrix 3.1, the scale matrix 3.2 and a rotation matrix around each axis in 3.3. The rotation matrix M_{ry}, for example, rotates a vertex v by degree α around the y-axis. This becomes clear by looking at element $M_{ry}[1][1]$, which signifies that all resulting values will lie on a xz-plane with $y = v.y$. We emphasize that the presented rotations rotate around a specific axis of the coordinate system. That means the coordinate system itself is rotated.

$$M_t = \begin{vmatrix} 0 & 0 & 0 & T_x \\ 0 & 0 & 0 & T_y \\ 0 & 0 & 0 & T_z \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{3.1}$$

Translation components.

$$M_s = \begin{vmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & S_g \end{vmatrix} \tag{3.2}$$

Scale components.

$$M_{rx} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha & 0 \\ 0 & sin\alpha & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad M_{ry} = \begin{vmatrix} cos\alpha & 0 & sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\alpha & 0 & cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \quad M_{rz} = \begin{vmatrix} cos\alpha & -sin\alpha & 0 & 0 \\ sin\alpha & cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{3.3}$$

Rotation components for each axis, x, y and z.

Unfortunately, matrices are not commutative. This can easily be verified by considering a vertex v on the y-axis with $v.y \neq 0$. If we rotate now with M_{ry} by some degrees v will not change. But if v is rotated beforehand by 90 degree with M_{rx} the operation M_{ry} with $\alpha \neq 0$ will indeed rotate v.

Multiple matrix operations can be concatenated by multiplying the separate matrices to a matrix M_c. The order matrices occure is important since matrix multiplications are not commutative. So, if M_c is the result of matrices multiplied in the order their operations occur, then M_c is used on the right side ($v\prime = v * M_c$), and vice versa. Suppose there are several vertices to be rotated by M_{rx} and M_{ry} and eventually transposed by M_t (from equations 3.3). We can either multiply each vertex with all matrices separately or multiply each vertex with the concatenated matrix $M_c = M_t * M_{ry} * M_{rx}$. And because the order is reversed, M_c is used on the left side ($v\prime = M_c * v$). The importance of this feature becomes apparent by the end of this chapter.

The rotation matrices in equation 3.3 are denominated **Euler rotations**. They bear the danger, in particular when they are consecutively concatenated, of eliminating the freedom to rotate around one of the axis. That happens when two axes align with each other or become very close to it. Imagine, in a right-handed coordinate system, a 90 degree rotation around the x axis, aligning the z axis with the y axis. A subsequent rotation around the z axis would not be distinguishable from a rotation around the y axis. Such a state is known as **gimbal lock**[Vin06, p. 73].

To prevent a gimbal lock, one may rotate around an arbitrary axis instead of each of the 3 axes separately. Equation 3.4 shows the matrix M_a to rotate α degree around axis through the origin and $v = (a, b, c)$. Its derivation is not a simple concatenation of the matrices 3.1-3.3 and outside the scope of this document.

$$M_a = \begin{vmatrix} a^2K + \cos\alpha & abK - c\,sin\alpha & acK + b\,sin\alpha & 0 \\ abK + c\,sin\alpha & b^2K + cos\alpha & bcK - a\,sin\alpha & 0 \\ acK - b\,sin\alpha & bcK + a\,sin\alpha & c^2K + cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \tag{3.4}$$

$$\text{with K} = 1 - cos\alpha$$

$$\text{and rotation axis v} = (a, b, c)$$

Rotation matrix around an arbitrary vector v by α degrees.[Vin06, p. 79]

We discuss in section 3.2 a more performant way to rotate objects.

Matrices are used to transform objects from one coordinate system to another. That brings us back to our initial topic.

Vertices are rarely considered as single entity, but rather grouped to form a **model** (alternatively called an **object**). Such a model resides in its own space called **model** space. For example, a model may shape a car, a tree or only a leaf, depending on how it can be reused. The main idea is to construct a **scene** out of such models.

A scene lives in **global** or **world** space and contains all objects, which may be rendered on the screen. Assembling a scene means to *copy* transformed versions of objects from their object space into global space. Copying and transforming is achieved by multiplying each vertex of an object with a matrix. Matrices for that purpose are called **object** matrix and follow the construction we discussed earlier. Each object has it's own object matrix, or even several object matrices, to create a new instance of an object with matrix-specific transfigurations. Hence, a single tree model can be used to construct a forest where each tree instance looks different due to the tree instance-dependent model matrix. This fact is depicted at the beginning of figure 3.2. There, we show the whole transformation process of a model until it is visible on the screen.

So, assuming that the global space is filled with the desired objects to visualize, a fraction of the global space can be projected onto a 2D window on the computer screen. The window receiving the projection is represented in global space by a rectangle called **viewport**. Its origin is defined at the global space's origin. It has its view direction along the negative z-axis and an up vector along the positive y-axis. While the dimension of the viewport in x and y direction must be defined by the application, its location and

direction is stationary.[2]

Starting at the viewport, the **view volume**, which we discuss shortly, spreads out in view direction with a user-defined length. Only objects inside this volume are candidates for rendering on the screen. We say candidates because an object might be occluded by an object closer to the viewport and therefore not rendered.

Because the viewport is at a fixed location, it is up to the developer to simulate movement through the 3D world. This movement is achieved through repositioning the world's inventory by multiplying each vertex present in global space with a **view matrix**. Afterwards, are all objects situated in **view space**. The view space's origin is the point of view towards projection occurs. That makes the transformation into view space analogous to placing a camera in the real world; marking the spot and determining the view direction. Hence, the view space is often refered to as **camera space**. We refine the term camera later in this chapter. So, when the user wants to rotate his view by 30 degrees to the left, all objects in world space are rotated around the camera's current world position by 30 degrees to the right. A view matrix equals therefore: the camera's negated rotation-angles translated by the camera's current world position.

A view matrix is shown in equation 3.5 and can be interpreted as follows: The last row determines the negated position of the camera in world space. The negation puts the world coordinates relative to the camera's origin. The rotation components *right* and *up* define where the right side and top of the viewport are, while *lookAt* expresses the direction of the point of interest. A common initial setting is *the look at direction* set to $+z$, *up* to $+y$ and *right* to $+x$. This builds another space where the columns of the rotation component build the new x, y and z axis.

$$M_v = \begin{vmatrix} right_x & up_x & lookAt_x & 0 \\ right_y & up_y & lookAt_y & 0 \\ right_z & up_z & lookAt_z & 0 \\ -pos_x & -pos_y & -pos_z & 1 \end{vmatrix} \quad (3.5)$$

A View Matrix.

Figure 3.2 shows the view space together with the two common view volumes. The view volume, positioned and oriented through the view matrix, determines the cut-out of the scene, containig the objects to be rendered. Objects outside of the volume are culled, while objects crossing the volume's edge are clipped and displayed partially on the screen's edges. This prevents unecessary processing of object data outside the view volume.

The shape of the view volume determines how the vertices are projected on the screen. It is usually a cuboid, in case of **orthographic projection**, or a frustum in case of **perspective projection**. The visual differences of the projections are depicted in figure 3.3. Orthographically projected objects are projected without taking their distance to the camera into account. Hence, two objects with the same dimensions but different distances to the viewport will be drawn with equal size on the screen. With perspective projection, closer objects appear larger on the screen while remote objects are displayed smaller. Perspective projection imitates the view we know from our daily lives. Since we are particularly interested in presenting state charts spatially, we do not discuss orthographic projection further.

[2]The viewport configuration depends on the graphics library, in this case OpenGL.

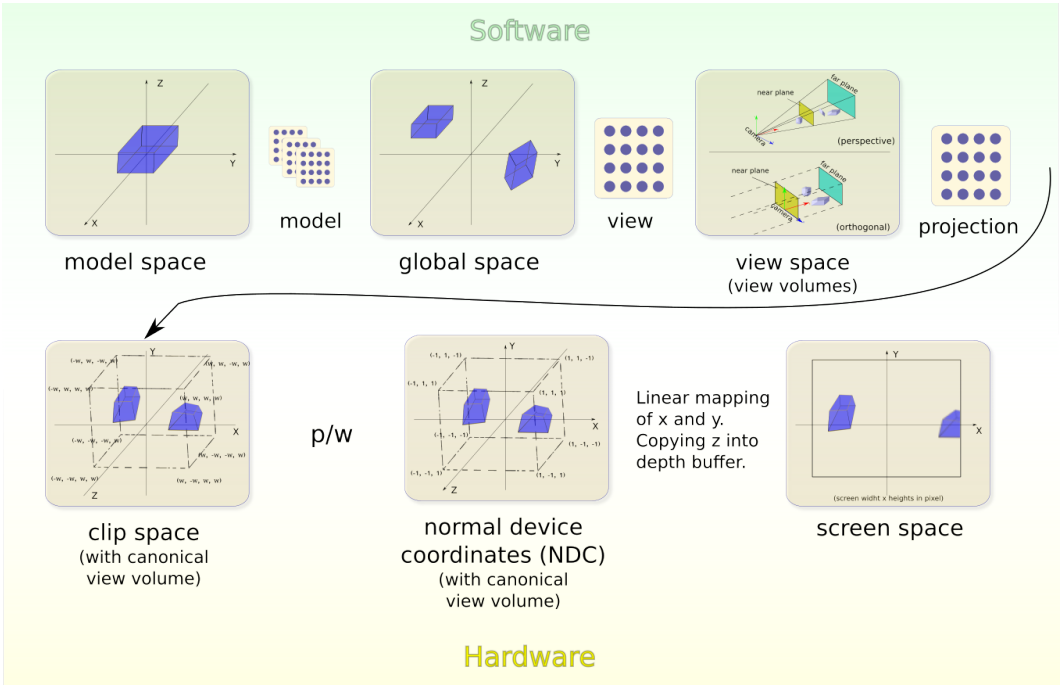


Figure 3.2: Spaces and matrices used to map between spaces. Common graphics hardware expect the software to provide clip space coordinates which are then transformed into screen space coordinates.

The **projection matrix** (PM) is constructed from the view volume's parameters. It is used to transform all vertices from view space into **clip space**. This transformation occurs in a way that the view volume is scaled into a cube; with its centroid lying in the origin and with its edges having a length of $2w$ in clip space. The newly shaped view volume is denominated **canonical view volume** (CVV) and depicted in figure 3.2 (clip space). Several goals are accomplished with this procedure. They are best explained by transforming a frustum into a perspective projection matrix (PPM).

Figure 3.4 shows a pyramid in view space. The *eye* on top of the pyramid indicates the position given by the view matrix. It represents also the beginning of the *line of sight*, or **center of projection**. The line of sight passes through the centers of the near- and far plane.[3] The view matrix's up-vector is aligned with the Y-vector. A frustum marks the view volume inside the pyramid. It consists of the 6 planes, **top**, **bottom**, **right**, **left**, **near** and **far**. The near plane represents the viewport, and is the target for the projection. It expands in clip space by 1 in x and y direction. That means the application's window width and height is equivalent to 2 in clip space coordinates, which are also called **normal device coordinates** (NDC). Expanding the near plane infinitly in x and y direction leads to the **projection plane**.

The perspective projection matrix is constructed from the frustum's parameters. To understand the PPM, we take its components from equation 3.6 and discuss their objectives as well as their development.

[3]This is true for symmetric frustums only. Asymmetric frustums[AMHH08, p. 836], which are used in stereo viewing are excluded from the discussion.

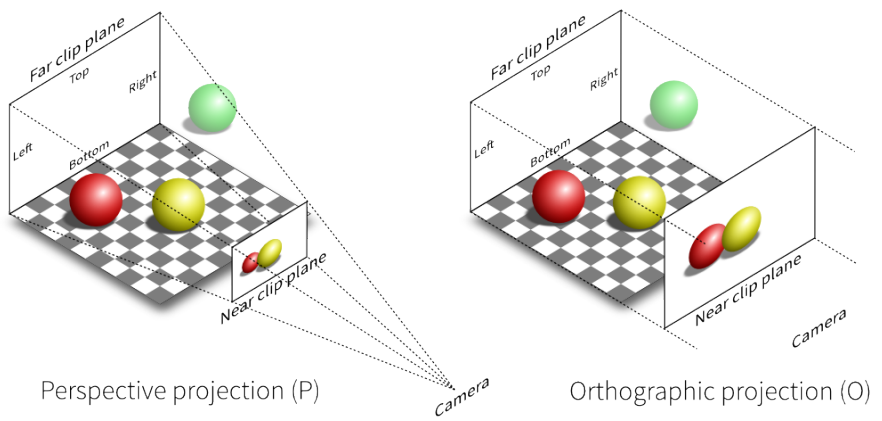


Figure 3.3: Perspective and orthographic projection.[Rou14]

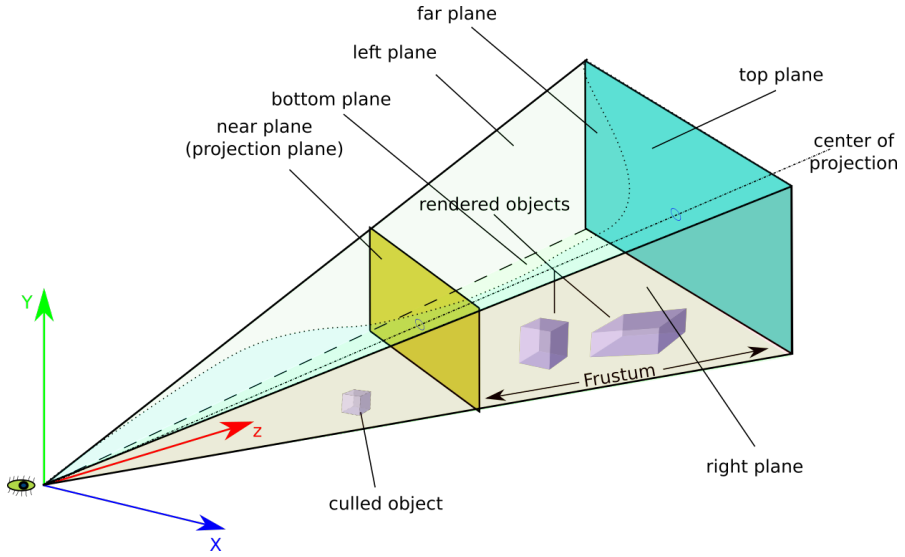


Figure 3.4: The frustum determines a cut-out of the global space which is to be perspectively projected onto the near plane.

When an object is projected, a zoom in x and y direction occurs to compensate for non-quadratic view planes. Without this correction, the displayed objects would appear stretched in either direction. The zoom factors are equivalent to scale factors and take the positions in PPM according to equation 3.2. Their values can be determined with the dimensions from corresponding planes of the frustum:

$$PPM = \begin{vmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & 1 \\ 0 & 0 & -\frac{2fn}{f-n} & 0 \end{vmatrix} \tag{3.6}$$

Perspective projection matrix, with distances to **n**ear, **f**ar, **r**ight and **t**op plane.

$$z_x = \frac{near}{right} \qquad z_y = \frac{near}{top} \tag{3.7}$$

In addition, the *physical* pixel of a display may be elongated and is therefore included when determining the zoom values. The ratio between the zooms is expressed as *aspect*

ratio[4]:

$$ratio = \frac{z_x}{z_y} = \frac{near\ plane\ width\ *\ pixel_{phy}\ width}{near\ plane\ hight\ *\ pixel_{phy}\ height} \tag{3.8}$$

Sometimes it is more desireable that the zoom factors depend on the angle between left and right (z_x) or top and bottom (z_y) planes. Since the determination of z_x and z_y is alike, we show how z_x is derived from the frustum, while the same is applicable for z_y.

This angle is called **field of view** (FOV) and it satisfies: $z_x = \frac{1}{\tan(\frac{1}{2}fov)}$ [DP11, p.367].

Figure 3.5 illustrates two horizontal FOVs in camera space. The blue line, expanding by 1 in the $-x$ and x directions, represents the viewport. The content of the viewport is represented by the gnu. A large gnu is depicted on the left side where the FOV has an angle of 30°. On the right side of the figure the FOV is increased to 60°, letting the near plane catche a wider range of view space. Since the width of the window did not change, the number of pixels is still the same. That means, a larger part of the view space must now fit into the same amount of pixels. From this follows that each pixel catches a larger portion of the view space, which induces a smaller gnu.

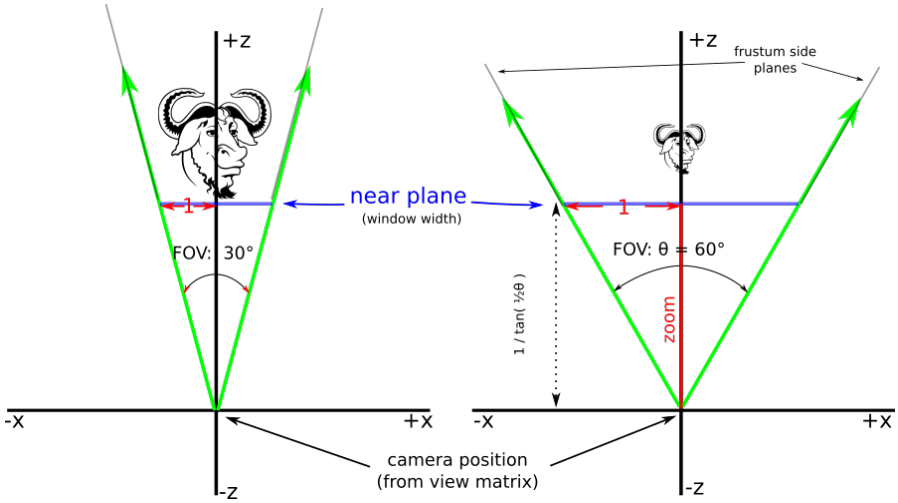


Figure 3.5: A 2-dimensional *field of view* (FOV) in camera space. Since the near plane's geometry is fixed, increasing the angle means more global space must be projected to the same area.
This results in less space for the gnu, ergo, a smaller gnu[Hec, picture] is displayed.

The calculated z value is the length of a normal between the vertex to project and a plane spannig through the camera. It is biased in such a way that if z lies on the **n**ear plane, $\frac{z_n}{w}$ equals -1, and if z lies on the **f**ar plane, $\frac{z_f}{w}$ equals 1. A common denomination is depth value, since it represents the depth from the camera into the space in view direction. The depth value is used to determine which object is occluded by another.

The remaining component to discuss is the 1 in the 4th row in the PPM. When a vertex is multiplied with the PPM, this 1 copies the z-coordinate from the vertex into

[4]not accounted for in equation 3.6.

the perspective component w. This values is used later to calculate the perspective foreshortening.

With the projection matrix at hand, we can now finalize our camera analogy. While the view matrix is responsible for position and orientation, the projection matrix determines the characteristics of projection.

Since the frustum is transformed into a cube, all objects have been stretched in x and y direction, and the subsequent operations are equivalent to orthographic projection.

Until now, we have discussed how matrices are used to transform vertices from one space to another. Even vertices behind the camera, which are not displayed, are transformed into view space and eventually projected by the projection matrix into **clip space**. To determine if a vertex lies inside the viewing volume or outside of it, each of the coordinates x, y and z is tested against the vertex's homogeneous coordinate w. Vertices inside the volume have only x, y and z components in the interval $[-w, w]$. Vertices outside the volume are discarded and the object owning the vertex is clipped or culled.[SA12, p. 310]

The resulting set of vertices is mapped back from homogeneous coordinates to euclidean coordinates by dividing each component through w. This **perspective division** leaves the vertices in **normal device coordinates** (NDC). They have the range [-1,1] on all axes. Since the z value represents the depth of a vertex, it cannot be displayed on a two dimensional screen. On graphic adapters, it is therefore copied into the depth buffer, which is topic of chapter 8.6.1.

The remaining (x,y)-tuples are assigned to pixel coordinates, a transformation called *viewport transform*. Its position P is determined by:

$$P_{(x,y)} = (\ p_x = \frac{2 * x}{width}, \ p_y = \frac{2 * y}{height} \ ) \tag{3.9}$$

How these pixels are colored, specifically with OpenGL, is the matter of chapter 8.5.3.

In this chapter, the mathematical background to project single points from 3D space onto a 2-dimensional plane was given. Particularly, it was shown how matrices facilitate projection and rotation. But matrix rotations around an arbitrary axis is difficult to interpolate and concatenation of matrix rotations bears the danger of gimbal lock. There are several ways to avoid the gimbal lock. One of them is the use of a different number system named quaternion, which we discuss in the following chapter 3.2. Furthermore, a *camera* was constructed to parameterize the projection and to change the point of view in global space according to the user's desire. While this foundation is only partially necessary to understand the Blender implementation in chapter 4, it is extended when we talk about the OpenGL implementation in chapter 7.

3.2 Quaternions

Introduction

The proposed rotations in previous section 3.1 come with some restrictions. Euler rotations suffer from the unavoidable[AMHH08, p. 67] phenomenon gimbal lock, while matrix rotations, specifically formula 3.4 on page 26, cannot be interpolated easily[DP11, p. 237ff]. Quaternion is a number system, which facilitates rotation in 3-dimensional space, free from gimbal lock and easy to interpolate with a technique named slerp. Posts on various internet forums and the fact that many graphic books devote at least one chapter to quaternions indicate that quaternions are the method to chose when it comes to spacial rotations. Furthermore, Eberly analyzes in his white paper [Ebe02] different kinds of rotations and shows that quaternions are faster while consuming less memory than matrix rotations.

Our goal is to rotate a camera to an arbitrary point in space, which makes a rotation around all three axis at once necessary. A smooth movement, and therefore a linear interpolation of the rotation, is desirable to provide a fluid view through the camera.

The information in this chapter is mainly gathered from [Vin06], [AMHH08] and [DP11].

As a start, an introductory example gives an overview of how quaternion rotations are applied. Afterwards, an elaboration of the mathematical details and a basic geometric interpretation is given.

Euler's rotation theorem[DP11, p.244] says that any compilation of rotations of a point in $\mathbb{R}^3$ can be expressed by a single rotation around a specific vector. Quaternions do exactly that with the help of a 4th dimension; comparable but not identical with homogeneous coordinates.

Let's assume that a point p shall be rotated in $\mathbb{R}^3$ by angle α around the vector $\mathbf{v}$. Three quaternions are needed to perform such an operation. The first one is constructed out of $\mathbf{v}$ and α resulting in quaternion q_r. It needs to be normalized to $\hat{\mathbf{v}}$ to be used in a rotation. Building the inverse of q_r gives the second quaternion q_r^{-1}. This tuple, q_r and q_r^{-1}, stores the actual rotation and is equivalent to a 3x3 rotation matrix. The last quaternion is created by assigning p to the vector part of a quaternion q_p; the quaternion rotated. Its 4th component, the scalar s, is set to zero.

To finally apply the rotation to q_p, it is embedded into the **sandwich product** (SP) consisting of two quaternion multiplications: $q\prime = q_r q_p q_r^{-1}$. Multiplying two quaternions is denominated **quaternion product** and is a singularity in quaternion algebra explained later in this chapter.

The resulting quaternion $q\prime$ contains the rotated point. Transforming it back into $\mathbb{R}^3$ is easy since after applying SP, the 4th component s is always 0. It can therefore simply be dropped.

After this small overview of how points are rotated by quaternions, we discuss now the mathematical background.

Mathematical description

Quaternions are a number system operating in a Hamilton space[5] $\mathbb{H}$. It is derived from complex numbers; having a scalar value s and an imaginary part extended to a complex triple $i + j + k$. In addition to the expression $i = \sqrt{-1}$, the imaginary triple obeys the rules listed in 3.10.

$$\begin{gathered} i^2 = j^2 = k^2 = -1 \\ \begin{aligned} ij &= k, & jk &= i, & ki &= j \\ ji &= -k, & kj &= -i, & ik &= -j \end{aligned} \end{gathered} \tag{3.10}$$

In the form $x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$, with $x, y, z \; \varepsilon \; \mathbb{R}$, the triple can be mapped to a vector $v = (x, y, z) \; \varepsilon \; \mathbb{R}^3$. An arbitrary pointer, which is to be embedded into a quaternion, can rotated around this vector v. The angle of the rotation is reflected in the 4th element, the scalar s.

Both together, scalar and vector, form an element of $\mathbb{R}^4$; obeying all it's algebraic operations. That means adding, subtracting, multiplying a vector with a scalar, etc. is achieved as usual. By enriching the list of operations with the *quaternion product*, a way to multiply two vectors with each other, the definition of Hamilton space $\mathbb{H}$ and its elements, quaternions, is complete[Ago05, ch. 4.14].

Figure 3.6 shows the mapping of the complex axes into a Cartesian system and some of the imaginary products from equation 3.10.

[5] It is named to the honor of Sir William Rowan Hamilton (1806-1865), who discovered quaternions.

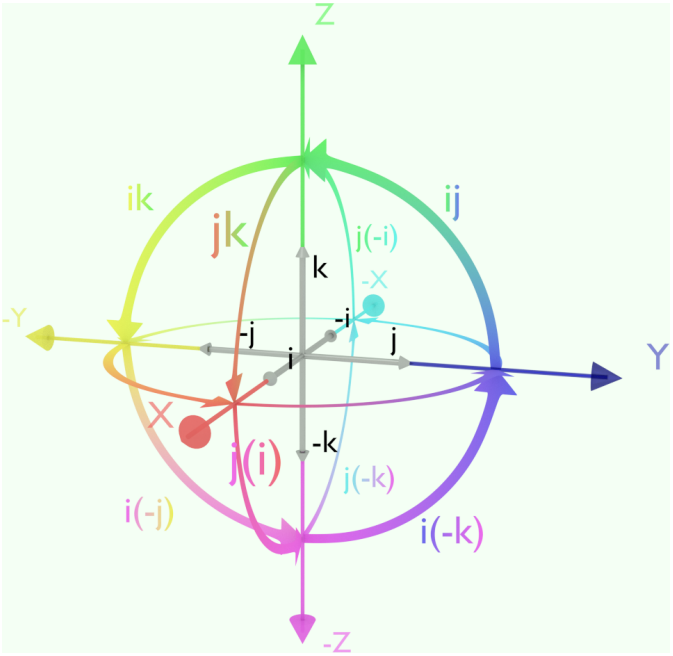


Figure 3.6: The Cartesian system mapped with the imaginary axes into it; arrows indicate some complex products. With $\hat{\mathbf{v}}$ an example vector to rotate around is given.

Analogous to complex number, quaternions can be expressed in different ways:

$$q = s + xi + yj + zk \tag{3.11}$$
$$q = s + \mathbf{v} \tag{3.12}$$
$$q = [s, \mathbf{v}] \tag{3.13}$$
$$\text{with } s, x, y, z \,\varepsilon\, \mathbb{R}, \quad \mathbf{v} \,\varepsilon\, \mathbb{R}^3$$
$$\text{complex components: } i, j, k \,\varepsilon\, \mathbb{I}$$

Equation 3.11 is known from complex numbers notation; a scalar s summed up with the imaginary parts i, j, k. The second form, 3.12 adds a scalar with a vector, and in 3.13, the quaternion is shown as an ordered pair. These definitions are interchangeable[Vin11, p.52]. But we use them to construct the unit-norm and the quaternion product, which eventually build the foundation for rotations and interpolated rotation.

Unit-norm quaternion

The norm of a quaternion is defined as: $|q| = \sqrt{s^2 + |\mathbf{v}|^2}$. Dividing the quaternion through its norm results in the **unit-norm**[6] quaternion $q\prime$ as shown in 3.14.

$$q\prime = \frac{q}{\sqrt{s^2 + |\mathbf{v}|^2}}; \qquad |q\prime| = 1 \tag{3.14}$$

[6]The term unit-quaternion is reserved for quaternions $q = [0, \hat{\mathbf{v}}]$ whilst $\hat{\mathbf{v}}$ has unit length.

A unit-norm quaternion is used to rotate another, arbitrary quaternion. Its vector is the one around which the rotation occurs. To prevent the result from being randomly scaled, the embedded vector needs to be a normalized.

Together with the relation $cos^2\theta + sin^2\theta = 1$, we can combine 3.12 and 3.14 to the half-angle representation 3.15.

$$q = cos\frac{1}{2}\theta + sin\frac{1}{2}\theta\hat{\mathbf{v}} \tag{3.15}$$

This formula encodes the angle θ into the vector $\hat{\mathbf{v}}$ as well as scalar s and is the reason why quaternions appear often mysterious and opaque. But it is in fact a smart pre-calculation, exploited by the quaternion product later.

In relation to the initial example, the quaternion q_r is constructed in this way.

Inverse quaternion

The inverse of a quaternion is created in the same manner as the inverse of a complex number. First, the conjugate is constructed with $q^* = [s, -\mathbf{v}]$, and then divided by q's magnitude's square as in formula 3.16. The division can be skipped for unit-norm quaternion, since they are already normalized.

$$q^{-1} = \frac{q^*}{|q|^2} \tag{3.16}$$

This, together with q_r, completes the quaternion-tuple to perform the SP.

Pure quaternion

A pure quaternion has a scalar $s = 0$ and the form:

$$q = [0, \mathbf{v}] \tag{3.17}$$

They can be rotated by the SP.

For our introductive example, p is stored into $q_p = [0, p]$.

Quaternion product

The actual rotation is performed by the quaternion product:

$$\begin{aligned} q_a q_b &= [s_a, \mathbf{a}][s_b, \mathbf{b}] \\ &= [s_a s_b - \mathbf{a}\cdot\mathbf{b}, s_a\mathbf{b} + s_b\mathbf{a} + \mathbf{a}\times\mathbf{b}] \end{aligned} \tag{3.18}$$

Its interpretation is not trivial. Goldman describes its geometric meaning as *enigma* and eventually deciphers it in [Gol11]. But that is far out of scope of this document. Nevertheless, we take a closer look at the SP, representing the complete rotation.

Rotation concatenation

Two unit-norm quaternions can be joined together with the **quaternion product**:

$$q_{rc} = q_2 q_1 \tag{3.19}$$

The resultant quaternion q_{rc} rotates than a point $\mathbf{p}$ as if $\mathbf{p}$ is first rotated by q_1 and than by q_2. Since quaternion multiplication is non-commutative, the order is important. Because of the associative property we can write:

$$q = q_2(q_1 \mathbf{p} q_1^{-1}) q_2^{-1} = (q_2 q_1)\mathbf{p}(q_2^{-1} q_1^{-1}) \tag{3.20}$$

While in theory the quaternion product of two unit-norm quaternion results also in a unit-norm, in practice, it is most likely that rounding errors occur. That makes it necessary to re-normalize the quaternion at the end of a concatenation.

Sandwich product

A complete rotation of a point p by an angle θ around a vector $\hat{v}$ is done with the sandwich product: qpq^{-1}. John Vince expanded this product as shown in formula 3.21 [Vin06, p. 99, 7.13]. It reveals a perennial scalar value of zero and an interesting term in the vector part with which the desired point $p\prime$ is calculated.

$$qpq^{-1} = [0, (1-\cos\theta)(\hat{\mathbf{v}} \cdot \mathbf{p})\hat{\mathbf{v}} + \cos\theta\, \mathbf{p} + \sin\theta\, \hat{\mathbf{v}} \times \mathbf{p}] \tag{3.21}$$

Based on that formula, we give a geometric interpretation. With the vector term split into three vectors, as colored in formula 3.21, the construction of the final position becomes visible. Figure 3.7 shows these three components and the point p rotated to $p\prime$ and $p\prime\prime$. Since all terms depend on sin and cos, two simple edge cases can be analyzed: With $\theta = 0°$ no rotation occurs and only the cosine-term (red) remains, hence p stays at p. Rotating p about $\theta = 90°$, the two terms blue and green determine the final position, which is $p\prime$. Beside the edge cases, a gold colored disc shows a rotation of $\theta = 45°$, resulting in $p\prime\prime$.

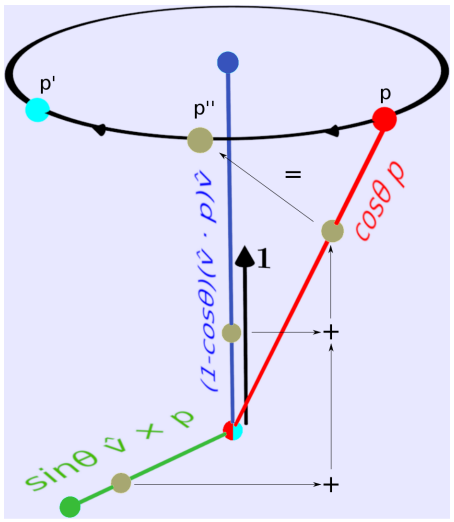


Figure 3.7: The sum of the separated terms of the SP for different angles (0°, 45°, 90°) result in the points on the circle.

Quaternion interpolation - SLERP

Rotation can be interpolated with **Spherical Linear Interpolation** (SLERP). This is useful to rotate cameras in a smooth way, not to confuse the audience. Details can be found at [DP11, p. 259 ff].

$$\mathbf{q} = \frac{\sin(1-t)\theta}{\sin\theta}\mathbf{q}_1 + \frac{\sin t\,\theta}{\sin\theta}\mathbf{q}_2 \qquad (3.22)$$
with θ as angle between the two vectors

3.3 Billboarding

Billboarding is a technique to rotate an object in a way that a specific side of it faces a target object. The adjusted object is usually a 2-dimensional polygon and is referred to as **billboard** (also called **impostor**). Its adapted orientation is often aligned with one or more axes of the target's local space. Anything can serve as target; a simple coordinate or another object. We focus merely onto the camera as target.

The billboard effect may or may not be noticed by the viewer. That depends on the quality[7] as well as the intention of this feature.

A common reason for using billboarding is to speed up rendering phenomenons consisting of tens of thousands of similar objects. Examples hereto are particle effects like smoke, fire, fog as well as vegetation, like grass or trees. Their elements are displayed as texture-quadruples. Each element's texture may be processed individually to let them appear unique.

Billboards can exploit the symmetry of objects. An isolated tree, for example, is cylindrically symmetric and appears the same from a distance, independent of any axial rotation of its trunk. Its complexity can therefore be reduced to a two dimensional image, billboarded to the camera, instead of having a complex, three dimensional polygon[MB05, p. 285]. There are also spherically symmetric objects, which are symmetric around two axes, like smoke particles or clouds. These are rotated around two axes.

Another reason for using billboarding is to keep objects aligned to the screen. Head-up displays (HUD) as well as annotations in space, for example, can be kept legible while the viewer moves around. There is a little distinction between these two applications. A HUD is drawn on the view-plane with a fixed depth value and can usually not be covered by other objects. This is different for annotations, which adapt their depth value and may be occluded by a scene's object.

[7] Some 3D-video games reveal this technique unintentionally. The monsters of the first person shooter Doom from the 90 are a well known example.

Akenine-Möller et al. describe several kinds of technique for billboarding [AMHH08, ch. 10.6].

Among others is the *world-oriented* billboarding. It comes in two variants illustrated in figure 3.8. The one to the left, *viewplane-oriented* billboarding, rotates all objects parallel to the viewplane. Since all objects are directed into the same direction, a single rotation matrix is sufficient for all billboards. The *viewpoint-oriented* approach, on the picture to the right, orients each objects separately to the camera position (viewpoint) and needs an individual rotation matrix for each object.

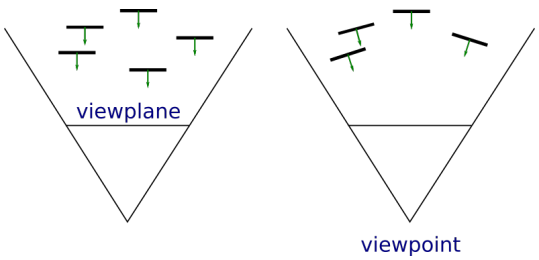
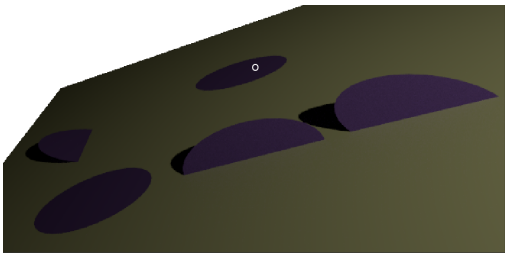
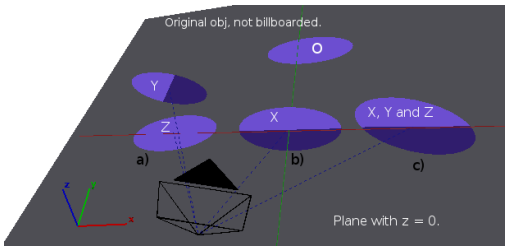


Figure 3.8: World oriented billboarding. Left viewplane aligned, right viewpoint aligned.



(a) Render of b) viewed through the camera. Note, a shadow indicates, that an object does not lie on the xy-plane anymore.



(b) Scene with objects aligned to different axes.

Figure 3.9: All ellipses were oriented as **o** in a). Through billboarding their orientation changed.

To demonstrate different alignment configurations, an ellipse **O** is billboarded in figure 10.5a around different axes, targeting the camera. For further clarification, the picture's scene is depicted in 10.5b with the billboarded axis noted. The dark blue areas of the ellipses mark the part underneath the xy-plane.

Billboarded objects can affect the three-dimensional perception when the camera moves. Contemplating the ellipse **a)** in 10.5b with a camera rotating counter-clockwise around the z-axis, the ellipse appears at some point aligned to the z-axis like **b**, but without the x-rotation. As the rotation goes further, **a)** will appear like **c)**, but with the z-rotation only. This effect is visible to the observant. In chapter 5.2, figure 5.3 gives a more realistic example.

Our objective is to have text labels as legible as possible. Furthermore, the user should not be distracted when text labels adapt their orientation during camera movement. Both can only be achieved by viewplane-oriented billboarding around all 3 axes. This is not mentioned in any of the references, since it is contrary to any natural appearance. For example, looking out of a rolling plane, one would expect that things move in the direction opposite to the planes movement, but with the method suggested, the world's objects rotate around their local center together with the plane.

Calculation

As precondition, we assume that every object to be billboarded starts with an initial orientation towards the camera. To determine the proper rotation matrix for billboarding, we consider figure 3.8 on the left. Imagine a clockwise (CW) camera rotation. All objects have then to be rotated counter-clockwise (CCW) to maintain alignment. This is achieved in vector algebra by applying the inverse of a rotation. This must be done for all 3 axes and leads therefore to the inverse of the camera rotation. Because the inverse of an orthogonal matrix is equivalent to its transpose we can just transpose the camera rotation to speed up calculation.

We come back to billboarding in the implementation for Blender in chapter 5.2 and for OpenGL in chapter 10.2.1.

Visual effect

In a common 3 dimensional environment, a forward movement results in a distortion of the elements on the border of the field of view[8]. By keeping the state-labels viewplane-aligned, they are prevented from being distorted. Since an observer expects the distortion, he may feel disoriented. To counter this effect, an artificial horizon could be implemented.

3.4 Algorithms

SCXML does not provide any facility to store positional indicators. Even with the extension we introduce in chapter 2.5.5 to restore a previously layed out stateroom, there are reasons one may want an automated, generated layout. That is, for exapmle, when an SCXML is loaded that was created with a different editor, not offering 3D positional information. Another reason is automated adapting to a differently sized display or even data imported from a different state machine description then SCXML.

When it comes to laying out graphs automatically in 2D or 3D, it seems that there are two popular methods. One is named *force-directed*[FR91] layout and the other one *simulated annealing*[DH96]. Both methods iteratively adjust the graph's vertices until a termination condition is reached. Fruchterman and Raingold (FR) conclude in [FR91] that their implementation of the force-directed method has 'interactive' speed, while Davidson and Harel state that their implemenation[DH96] of a *simulated annealing* method is focused on drawing a graph nicely and is not suited for 'interactivity'.

We do not rely on interactive layouting in our stateroom; adapting the graph dynamically after its topology changes is out of scope of this work. So, an initial setup time to draw

[8] For example when starship enterprise speeds up to warp, the light rays' length increases with the distance to the center.

the graph would be tolerable. The reason we use Fruchterman-Raingold anyhow, is still its speed, which is an advantage when the algorithm is tested over and over again.

The principle idea of the algorithm is to iteratively adjust the position of each vertex by applying two counter operating forces. Assuming a graph G consisting of vertices V and edges E and assuming further two functions, $f_r()$ and $f_a()$, which are used to apply the *repulsive* and *attractive* forces, respectively. Then, each v in V is displaced by adding the result of $f_r()$ applied on each distance v to each other element in V; repelling vertices from each other. Attractive forces displace each v, which is part of edge $e \in E$, with the result of $f_a()$ applied on the length of e; moving two vertices connected by an edge closer together.

The challenge to specify a successful algorithm is then to define the two functions calculating the forces. FR experimented successfully for 2-dimensional layouts with functions depending on a maximum available *area*, a maximum *size*, equally for all objects and a constant found expimentally.

In we use the 3D version of the FR algoritm provided by igraph[tea16].

Part III

Blender

Chapter 4

Introduction to Blender

Blender is a tool to work on computer graphics in several aspects. Among its wide range of functionalities, it offers the necessary facilities to build a stateroom editor: 3D object modeling, rendering, animating 3D models and video gaming. The quality of animations created in Blender was proven many times. Prominent projects are: Tears of Steel [Hub12], Cosmos Laundromat [Auv15] and Sintel [Lev10]. The environment inside Blender for the creation of games (BGE) is less sophisticated, but some results look promising [Art15].

This chapter gives first a small introduction into the BGE, while we refrain from explaining Blender itself. Afterwards, our attempt to create a *game* to design staterooms is documented. We show how the heads up display (HUD) and the movement is set up and how the models for the editor are constructed in Blender. Besides a lot of unsolved problems, we argue in particular that drawing a transition, as desired by a player, is difficult to achieve in BGE in its current state. We substantiate our findings by showing a working model in Blender and the problems to manipulate this model in BGE. We finally conclude that BGE is not suited to edit staterooms.

Nevertheless, Blender and BGE become useful in chapter 11 when the hardware of a watch is simulated and debugged.

Literature about BGE is rare. The few books available, [Chr11], [Bac12], [Fla10], give at best an introduction to the topic. Blender's Python API, an important aspect, is not covered in any of these books. Therefore, most of the information presented here was revealed by trial and error, discovered with the help of the community, specifically through the irc channel #blender (user Olson in particular) or learned from tutorials of various web pages. The irc channel #gameblender is unfortunately less supportive; probably because it is not well visited.

Chapter 5

Blender Game Engine

The BGE is a special mode of Blender. It allows to start a scene developed in Blender as game. Such a game consists roughly of models interacting with each other, a *world* in which the *game* actually happens and a set of rules called game-logic, defining what is allowed to happen.

In chapter 3.1 we introduced the foundation of models. Blender facilitates designing such models and offers additional features like textures, materials and equipping models with a physical description. To have an objects visible in another color than white a material must be assign to it. The material can than be colored in any thinkable way.

When a game starts, the current scene becomes the world environment or level. Such an environment can be fashioned by placing a landscape on the ground or putting stars in a distance. BGE offers a few effects to have a slightly more appealing world environment as the plane empty space.

Our focus lies on a prototype for a stateroom editor. For that reason we use only plain color materials to make states, transitions, etc. visible. There are no sun or spot lights, just an artificial ambient light, which is necessary to actually see something. We activate the world's mist feature so that objects fade out smoothly when reaching the far-plane. This looks nicer than having states disappear abruptly. A textured hyperplane in the middle of the room helps the user to orientate and gives visual feedback during movement. While there are many world effects one can think of, we don not explore this subject further.

The game-logic determines how models interact with each other and how the player's input affects the game. Blender's Logic Editor is a tool to design this logic graphically. The next section is devoted to that tool.

5.1 Logic Editor

Blender's logic editor[Com16a] (LE) is developed for people without programming skills. The principle idea is to have predefined logic bricks suited for a specific task and connect these bricks to the logic desired. A brick can be a sensor, a controller or an actuator. The general workflow is that a sensor triggers a controller, which then possibly triggers an actuator or executes some Python code. Figure 5.2 depicts the LE with three columns (1) for the bricks. Sensors are created in the first column. Their output anchor (2) can be connected to several controllers. A controller receiving a signal re-evaluates its state and, depending on its type, executes a Python script or sets its output value.

The *wires* in the figure may give the impression that the interaction works like electric current: If the sensor becomes active the controller is turned on and with it, all connected actuators. And if the sensor becomes inactive, the controller/actuator is turned off again. But the interaction is more complicated. A sensor becoming active emits pulses to the connected controllers which may then trigger actuators. These run afterwards independent from the sensor to its completion. This aspect and its consequences are refined later in the *Sensors* paragraph on page 46.

Besides logic bricks, the LE offers *Game Properties* (5) (GP). These properties can be a typed variable, like float or string, or a timer. For the presented work, only variables are considered.

Logic bricks are always associated with a Blender model. In figure 5.2, all bricks belong to the model *Quit*. Hence, bricks operate in the environment of that model. In addition, the BGE extends the environment by a simple state machine. It is basically a bitmask $\neq 0$. Each bit represents a state with all states being orthogonal to each other. If a controller belongs to the state 4, and the object is in state %011, the controller is disabled. How these states are managed is the described in the two paragraphs, *Controller* and *Actuator*, below.

A logic brick is divided into *common options* (CO) and *specific options* (SO), as shown in figure 5.2. The CO consists mostly of administrative properties like minimize, discard, name, type, etc. Sensors and controllers extend these CO, which are addressed in their sections.

The following paragraphs lists first the available logic bricks, while the discussion is limited to the ones used in this work, which are emphasized in **bold**. We starting with *Controller* bricks, continue with the *Actuator* and describe the *Sensors* facilitates last.

Controller

BGE offers the following controllers:
AND, **OR**, NAND, NOR, XOR, XNOR, Expression, **Python**

Figure 5.1: The controller column with states (blue rectangle).

The **Python** controller executes a Python script when receiving a pulse. A subsequent actuator is not possible[1]. All remaining controllers evaluate a logical expression upon the connected sensors and signal connected actuator bricks.

Controllers are assigned to a **state**. To be active, the model's internal state machine, with which the controller is associated, has to have the state enabled. Figure 5.1 shows how states are assigned to a controller. The blue rectangle in the CO section assigns state index 1 to the controller Python. Setting and resetting states is achieved through the Python API or through the State actuator. The yellow rectangle encompasses the *state panel*, which allows to show/hide the controller groups in the editor (Visibility) as well as set the initial value on start-up (Initial). Each Blender object supports up to 32 state indices.

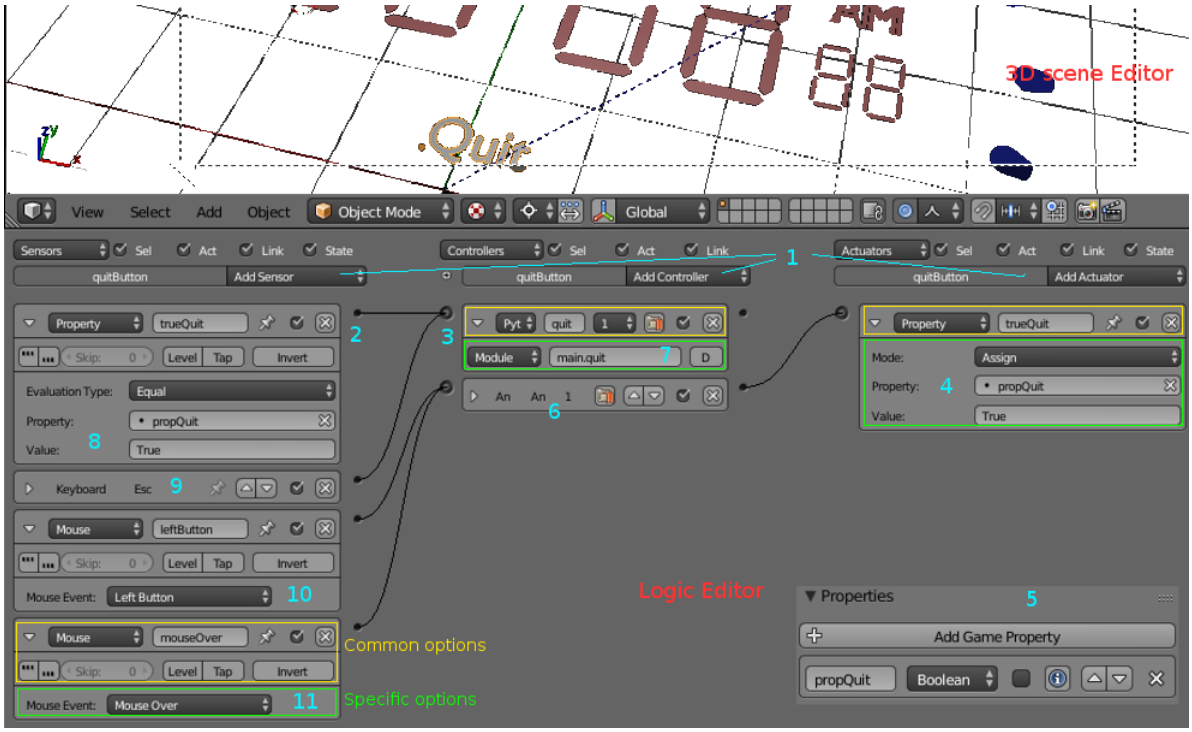


Figure 5.2: Executing a Python script on mouse click with logic bricks. The bricks are connected to the object *Quit*.

[1]This is probably a design flaw. It would be desirable to have the Python brick as actuator, so that the execution of a script depends on several sensors.

Actuator

BGE offers the following actuators:
Action, Camera, Constraints, **Edit Object**, Filter, **Game**, Message, Mouse, **Motion**, Parent, **Property**, Random, **Scene**, Steering, **Sound**, **State**, **Visibility**

Actuators describe how an object or the environment should be changed.

The **Property** actuator, number (4) in figure 5.1 changes a *game property* according to its mode. These modes are fairly self-explanatory, with names like assign, toggle or add. A change triggers all sensors watching this property, and make them reevaluate their status. We come back to this in the next section, when we talk about sensors.

The **Motion** actuator comes in two flavors: simple motion and servo control. We only use simple motion to transform an object by a specified translation and rotation. Servo control applies forces to mimic physics and the attached objects move smoothly through the space. This is important for collision detection.

With the **Edit Object** actuator, several modifications on objects can be done. Here, we focus only on two: **Add Object** and **Track to**. The former mode allows to copy an object into the scene, while the latter allows to billboard to another object. The billboarding feature is limited to axial or spherical billboarding, cp. chapter 3.3.

The **Visibility** actuator is used to set or unset the visibility of an object. It has two further options, one propagates its visibility state to its children and another to occlude the objects behind it.

In the previous section, we discussed the state mechanisms for controllers. The **State** actuator allows to manipulate the state of the state machine. By resetting a state, all corresponding group controllers are deactivated.

With the **Action** actuator the game engine can play animation sequences which is a nice feature to temporarily highlight specific objects.

The **Sound** actuator plays a chosen sound when activated.

Blender can manage several scenes at the same time. The **Scene** actuator allows to switch between scenes or put a scene as background resp. foreground (overlay).

Game is used, among other functions, to restart or end the game.

Sensors

BGE offers the following sensors:
Actuator, **Always**, Collision, **Delay**, Joystick, **Keyboard**, Message, **Mouse**, Near, **Property**, Radar, Random, Ray

Before discussing each of the used sensors, a short description of its CO, presented in figure 5.2, is given. Sensors activate controllers when a certain event happens. This

activation is done by a pulse which can be configured in the second line in the CO, in the figure marked as (11). By default, edge-triggering is active and only state changes from high to low or vice versa result in a pulse. The two 3-dotted buttons toggle level triggering, one toggles positive level triggering, the other negative level triggering. That means as long as a state is positive, resp. negative, the sensor keeps issuing pulses at a certain frequency. The pulse-rate on level triggering is by default the same as the frame-rate. This can be adjusted by the frequency option. It seems that level triggering works only for Python scripts and maybe for some actuators. We give an example of this in section Final words about Logic bricks on page 48.

There are three more switches in the sensors' CO. The **level** option informs a connected controller when the controller's state machine changes. This is valuable in case the sensor has already sent a pulse but the controller was deactivated by the state machine.

We do not engage into the **Tap** switch, which gives in general just an extra pulse[2]. The last switch **Invert** simply applies a logical not to the sensors' output.

The **Always** sensor and the **Delay** sensor trigger by default at the first frame and stick to true. While the **Always** sensor comes with no further options[3] than the common options, the **Delay** sensor allows to parameterize a duty cycle.

A **Keyboard** sensor can be programmed to send a pulse when a single key is pressed, like *a* or *b*, or any key at all. Modifiers, like shift or control, are also supported.

The **Mouse** sensor can be configured to react to one of the following mouse events: button pressed/released (left, right, middle), wheel up/down, movement and mouse over any. All these events are triggered independent of whether the mouse hovers over the related object or not. It is also not possible to detect if a mouse button was pressed or released.

Mouse over is an actuator, which leads to a pulse only if the mouse hovers over the object belonging to the brick. So, detecting if an object was clicked, this and the previous, mouse sensor, have to combined with an *and* controller.

With the **Property** sensor one can detect when a *game property* changes, respective changes to a specific value. This makes it an ideal candidate to listen to user-defined events, for example when a counter reaches a certain value.

The **Actuator** sensor observe actuators and triggers connected controllers whenever the actuator becomes active. An application for this is to trigger state-dependent controllers when a state change occurs.

[2]The additional pulse is given when the sensor changes its state from false to true, but with some special rules when it comes to level-triggering.

[3]That makes it actually superfluous because the functionality is also include in the Delay-sensor.

Example

A complete example how logic bricks interoperate is illustrated in figure 5.2 on page 45. It shows a facility to run the exit script *main.quit* (7) when either the escape key is pressed or the *Quit* object is clicked. The brick for the escape key is a straight forward *Keyboard*-sensor (9) connected with the controller for the script.

The object clicking needs two sensors, one to check if the mouse is actually clicked (10) and a second one to determine if the mouse hovers (11) over the *Quit* object. The *And* controller (6) ensures that both events take place at the same time before the *Property* actuator (4) sets *propQuit* to true. This setting is propagated to the *Property* sensor (8) which triggers eventually the Python script. Having a *Python* actuator would make things more easier and we could drop the Property here.

Final words about Logic bricks

At first glance, logic bricks seem to be easily understandable, but there are some phenomena or possibly bugs. It is hard to determine which since there is no specification. We give a small example to show how cumbersome the development with logic bricks can be.

Let a cube have an *always* sensor be connected over a simple *and* controller to a *motion* actuator. The *motion* actuator shall be configured with a simple movement of 0.01 into the X-direction. When the game starts, the cube moves constantly. But we would expect just a single step of 0.01 because level triggering is disabled. Activating level triggering and changing the sensor's COs skip option has no effect. According to the definition at BGE's manual, we expect a stop and go effect. If a script-controller with a Python script to print a simple message is hooked up to the sensor, the expected triggering can be observed.

Another issues with logic bricks is, that sensors and actuators cannot be connected directly, even though it is desirable in many cases. Using an or-controller as well as and-controller in-between works fine but gives the developer the burden of parsing this brick as pass-through controller.

Furthermore, it is a challenge for the developer, to keep track of which logic brick is responsible for what effect in the game. BGE tries to enforce some kind of order by coupling models with its controlling logic bricks. That seems to work in general, but fails in a lot of cases. Assuming a game is to be restarted, when the player falls down a cliff, time runs out, live points hit zero or the player presses restart on a menu. A solution with logic bricks could be to trigger a *game* actuator configured to restart the game. Each object detecting the restart event would have such an actuator. If the game to be extended, for example a dialog should pop-up to inform the player, than it would be difficult to find every logic brick causing a restart, since there is no search function for logic bricks. One may argue that a Python script would be more suitable location in the first place. But that just brings us to the last drawback we want to discuss here.

Python logic bricks cannot be backtraced in BGE itself. There is no way to have a list showing which objects call a specific Python script. A running script though, may detect the name of the calling controller, but that name is not related to the object the Python controller is associated with. So, changing a Python script may have unforeseeable consequences, if the script is called from different locations.

Python API

Blender has two APIs to Python, one for itself and a second one to BGE. They are very different from each other but have also some common functions, which makes it easy to become confused. The one for Blender is to control Blender itself through Python. That means drawing and editing meshes, or any other objects, can be done using either Blender's GUI or by calling the internal Python functions representing that GUI element.

The other API, for BGE, offers access to the internals of logic bricks. For example, a mouse click consists of a press and release event. Logic bricks themselves cannot differentiate between these two. But the Python API offers low level access to the controller, where this information can be read.

Because the BGE Python API is only accessible when the game is running, testing can be very cumbersome, in particular where the documentation is sparse or unclear. Some usage is shown later in chapter 6 where the stateroom-editor is implemented.

We omit any deeper API description since it would not contribute to the stateroom editor.

Layers

The BGE does not support creation of models or meshes. It can only copy existing models. A direct consequence is that every object that can possibly appear on the screen must exists in the game no matter when and if it appears. Blender supports 20 layers (or separate global spaces). So, one of theses layers can serve as *material store*, from which objects are copied into the scene. Libraries are also possible in Blender, but we do not investigate them.

When a model is copied, all logic bricks are also copied. There are no global or shared properties. Communication could be done via the *Message* actuator/sensor, but we excluded these from the discussion.

5.2 Billboarding

General billboarding was discussed in chapter 3.3. BGE offers this functionality under the designation *Track to* as option of the previous mentioned *Edit Object* actuator. Figure 5.3 shows the actuator tracking the *Camera* object. The local direction of a state label is the **-Y** direction, as shown in figure 6.4 on page 54. This option is called **Track Axis** in the actuator and selects the axis facing the camera. The **Up Axis** determines which local axis is turned up. With the **3D** button one can choose if the object is to be billboarded around the 3rd axis, in this case the **X** axis, too. Choosing a time value leads to an interpolated adjustment of the orientation over the period specified.

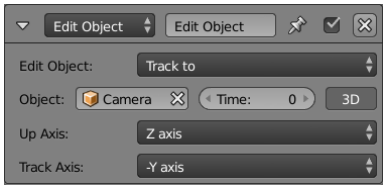


Figure 5.3: Actuator to billboard an object.

Because there is no *Track to destination axis* option, only viewpoint billboarding is supported; making screen-alignment, our requirement from chapter 3.3, impossible Figure 5.4 illustrate the problem. By translating the camera to the right, **S6** moves to the left and rotates in the orientation of **S7**. Imagine a screen filled with states, spinning like this, would distract an observer.

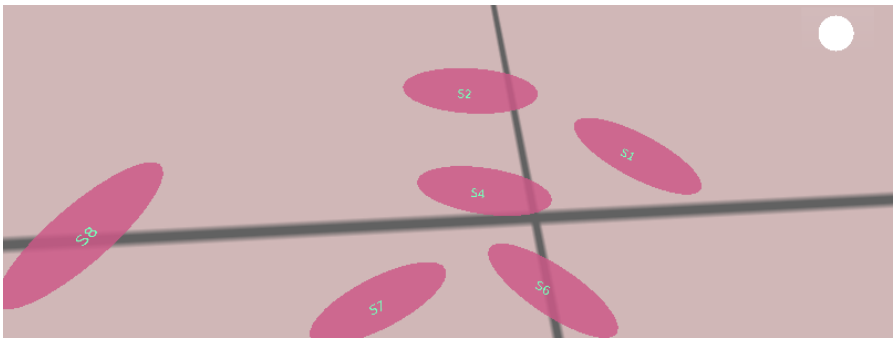


Figure 5.4: Looking down to spherically billboarded states.

Nevertheless, objects which shall stay always screen aligned in a certain distance to the camera can use the parent-child option of Blender, which works also in BGE. Through this relation all transformations applied to the parent are automatically propagated to its children. This is useful for menus or other HUD elements.

Chapter 6

Trial and error: Stateroom editor as game

In this chapter we illustrate our trial to implement the stateroom editor as Blender game. We focus first on the main building blocks to create a minimal stateroom, like movement, test, atomic state and transition. With the difficulties of implementing a transition, we finally abandoned this part of the project. At the end of this chapter, in section 6.4, we list unsolved problems, which piled up during development.

6.1 HUD - Heads up display

The heads up display (HUD), see figure 6.1, consists of a single button, *spawn*-button (top corner) and a textual menu (not visible on this picture). These elements are positioned very close to the camera to prevent them from being occluded by state-objects while the user wanders about. A better way to implement a HUD would be to construct it in a separate scene and use that as overlay with the **Scene** actuator. But since scenes do not share logic bricks or objects, new spawned states would appear in the HUD-scene and eventually defeating the whole purpose an extra scene. There is a chance that this would work with libraries, but we do not pursue this matter.

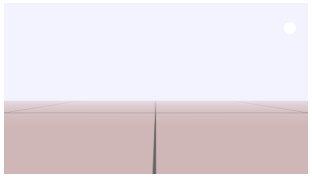


Figure 6.1: Empty stateroom.

Game Logic

Figure 6.2 is a screenshot of Blender where the game is not running. It displays the cut-out that the camera captures (screen-space) when the game starts, plus some space not captured. This non-captured space is colored dark gray and contains a textual menu. A Keyboard sensor can trigger an Action actuator to start an animation that moves the menu smoothly into the view-space. No further functionality is implemented.

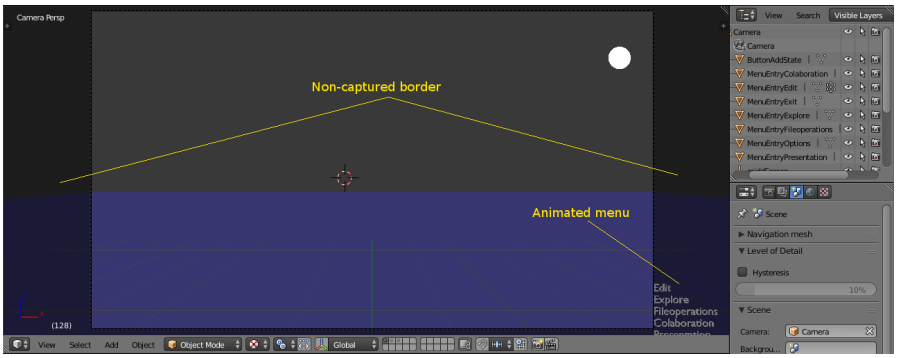


Figure 6.2: Hud during development, the spawn-button looks like a white moon. (Game engine is not running)

The spawn-button spawns new states when clicked upon and its position raises two problems. When it is pressed, a new state appears in front of the camera but behind the button. In other words, it occludes the just spawned state. If the mouse is than moved while its button is still pressed, the logic to spawn a new state would be triggered over and over again. A simplest solution for this is to hide the spawn-button when it is clicked until the mouse-button is released again. The corresponding logic bricks can be viewed in figure 6.3. We refrain from a detailed explanation since logic bricks are already explained.

Besides toggling the button's visibility, this construction calls the Python function *createState*, which copies a new state with a specific distance to the camera under the pointer's position.

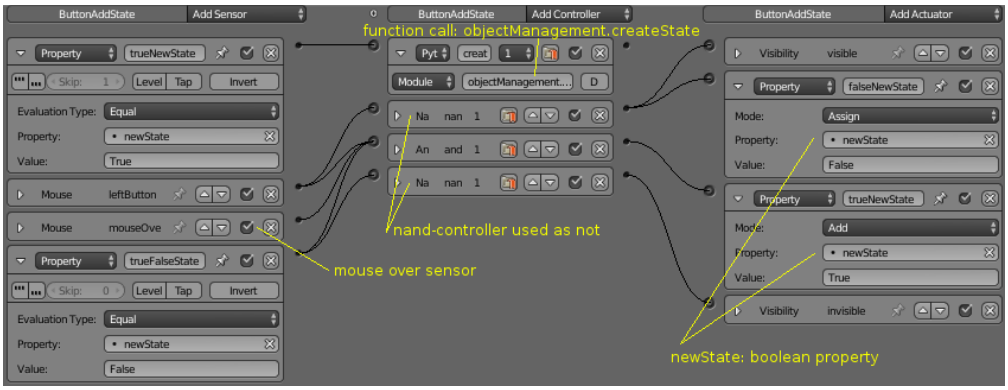


Figure 6.3: To prevent spawning uncontrolled states, the spawn-button disappears when pressed and reappears when the mouse is released.

6.1.1 Movement

There are many ways to navigate through space. We decided to have the camera move freely in all directions, with translation steered by keyboard and rotation by mouse.

The logic bricks for navigation are bound directly to the camera and navigate therefore in relation to it. That differs from the navigation in a first person shooter. There, movement is rather related to the body, with the camera on the height of a head, which results in a more realistic effect when the protagonist runs through the levels. Since we focus on staterooms, we refrain from a deeper discussion.

Translating the camera can easily be achieved by connecting *keyboard* sensors with *movement* actuators. If, for example, the *w* key is pressed, the camera moves forward in its local space until the key is released again. The orientation adjustment for the camera, triggered through mouse movement is more cumbersome. That is because there is no sensor to detect in what direction the mouse moved and there is also no way to set the position of the mouse pointer. Imagine an invisible mouse pointer and a camera rotation to the right when the mouse is moved to the right. The system does update the mouse pointer's position, despite the invisibility of the mouse pointer. At the point the mouse pointer hits the screen boundary the rotation would stop, magically from the player point of view. This makes is necessary to reset the mouse's position every frame.

The solution for this is the Python script in listing 6.1.[1] It first checks whether the system is in the *explore* state, the state in which the user moves around with an invisible mouse pointer. Afterwards, it assumes, that the mouse pointer started travelling at window position (100,100). That is because BGE cannot deal with negative mouse coordinates, which may occur when the mouse pointer moves outside of the BGE window. So, whenever the user enters the explore state, the mouse position is set hardcoded to (100,100) (not in the listing). The current pointer position is available in the mouse sensor named *Movement* (line 6). It can be accessed through the Python controller (line 5), which is calling the script. Since the owner of the calling controller is the camera, its global position can be retrieved as in line 11. With the travelled distance, calculated in line 8 and 9, the orientation is adjusted in line 12-14. The fact that we adjust z and x shows that rolling cannot occur. At last, we reset the mouse pointer position in line 16.

```
def movement():                                                    1
  if not stateMachine.isState( stateMachine.EXPLORE ):
    return                                                         3

  controller  = logic.getCurrentController()                       5
  mouse       = controller.sensors['Movement']
                                                                   7
  xpos = mouse.position[0] - 100
  ypos = mouse.position[1] - 100                                   9

  ori = owner.worldOrientation.to_euler()                          11
  ori.z -= ( xpos * mouseMovementSpeed )
  ori.x -= ( ypos * mouseMovementSpeed )                           13
  owner.worldOrientation = ori
                                                                   15
  render.setMousePosition( 100, 100 )
```

Listing 6.1: Python snippet to update camera orientation on mouse movement.

[1] Blender 2.77 offers now a mouse actuator to overcome this issue.

6.2 Modeling Atomic State

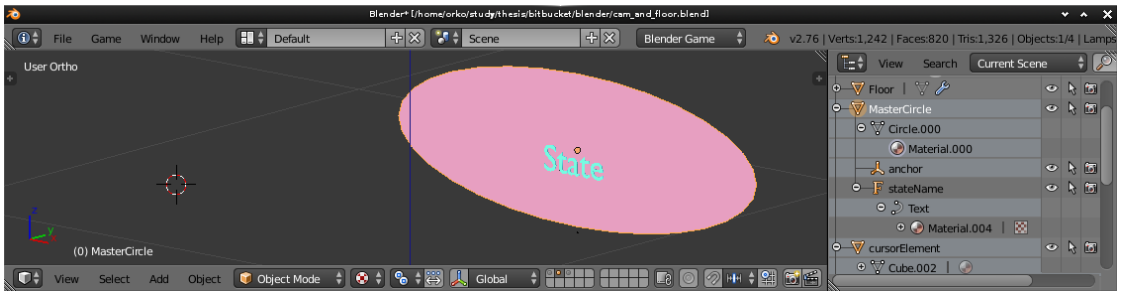
Figure 6.4: With 32 vertices is the prototype of the atomic state constructed.

The model representing an atomic state consists of a mesh formed as an ellipse (fig. 6.4), a text label and an anchor point. Figure 6.5a shows the model rendered, with its components listed on the right and the game logic at the bottom. The anchor point is a non-visible entity, a fixture in local space of the model. It serves as end point for transitions. In a finished version, this point should be recalculated for each transition to be on the border of the mesh, so that an arrow can attach the outline of the state. This problem is discussed deeper in the OpenGL part in chapter 9.0.8.

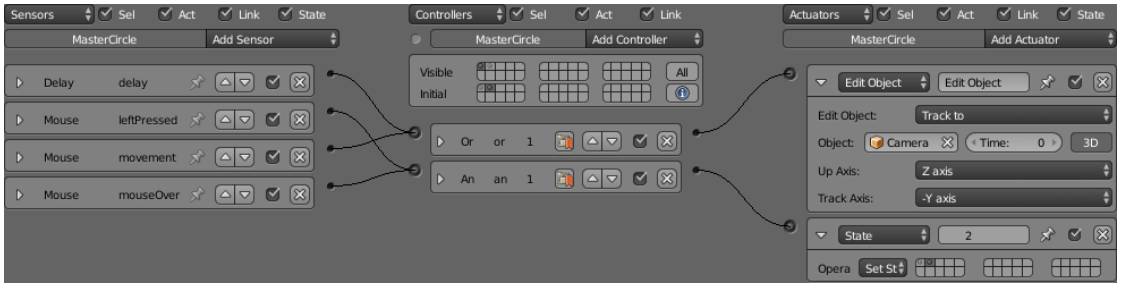
Game Logic

The logic is displayed in figure 6.5. When a state is spawned, its orientation is not billboarded. To correct this, the delay sensor triggers the billboarding actuator. It is also triggered whenever the mouse moves. This could be further optimized and only called when the state itself or the camera moves.

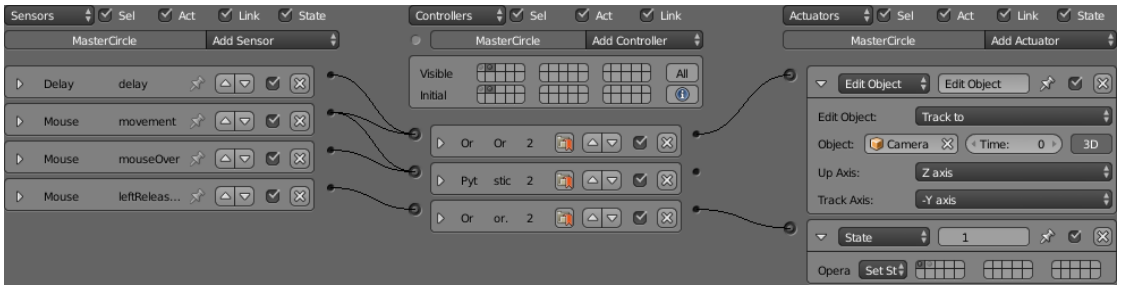
With the remaining logic a state can be clicked upon and relocated. Unfortunately, if during the relocation the mouse pointer hovers over another state, it will also be affected by the relocation process. Since BGE does not provide global variables and the logic bricks cannot differentiate between mouse press and release events, the bug can only be fixed via Python.



(a) The atomic state model. It's components are listed to the right under *MasterCircle* while the game logic is presented on the bottom half.



(b) Logic for state 1. While it awaits a click on the object to switch to state 2, it billobard to the camera.



(c) Logic for state 2. With the mouse button still pressed, a Python script is called, which makes the object follow the mouse pointer.

Figure 6.5: The atomic state and its logic bricks.

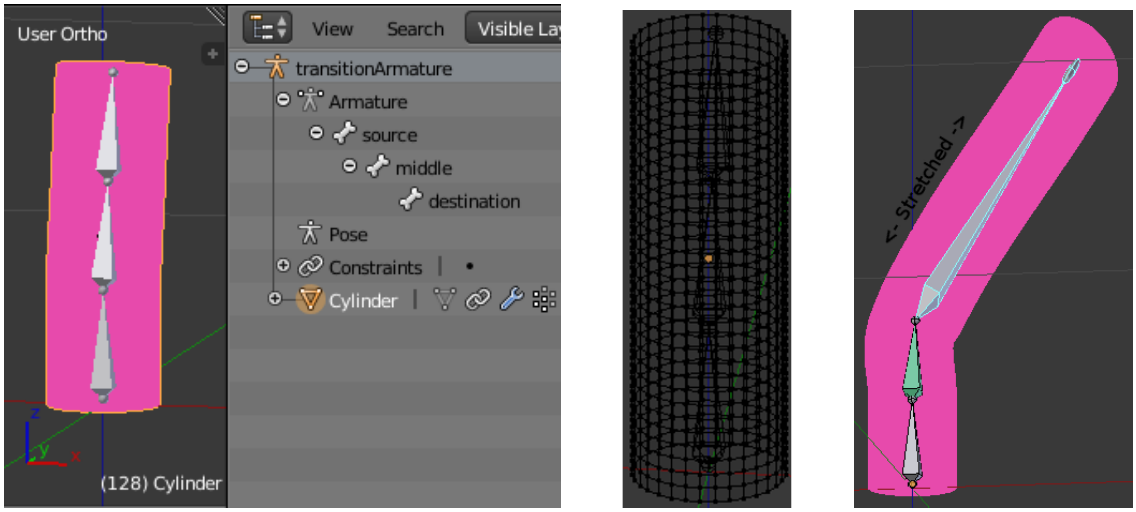
6.3 Modeling Transitions

A simple transition connects two end-points with a line. This line can be a billboarded 2D-rectangle or a 3D-tube. Achieving one of these solutions is mainly, from the point of view of BGE, the same task; reposition individual vertices from a mesh.

Assuming a quadrangle with 4 vertices shall be billboarded to form a transition. Because meshes cannot be generated in BGE, see paragraph *Layers* on page 49, the mesh of the quadrangle is placed on a non-visible layer. To form the transition, the mesh is copied to the visual layer. Since the user can place the states which are linked by a transition freely, the form of the quadrangle must adapted its dimension and location. For a first trial we choose the anchors as end points as described in the model in section 6.2 instead of calculating the points on the outline of the state.

Blender offers several methods to draw elements suitable for transitions, but unfortunately, they either do not work in BGE or are hard hard to maintain. We tried the following approaches:

- Building a transition out of bones. The principle idea is to have a chain of bones, like in figure 6.6. Scaling a bone in one direction leads to stretching a connected mesh. While translation and rotation do work in BGE, scaling does not.
- Deformation of volumetric objects works in both, Blender and BGE. Unfortunately, deformation cannot be controlled and does therefore not solve the task.
- Laying out transitions as splines does work in Blender, but again, not in BGE.
- According to BGE Python API, a mesh can be deformed through the controller, which we positively verified. Nevertheless, trying to access the mesh of a newly copied object failed. It is probably be possible to assemble a transition out of several short cylinders, forming a nice bezier curve. We consider this as to cumbersome since every piece has to recalculated everytime the camera rotates or moves.
- Considering a transtion as a *rope*, we found out that ropes are often animated through a *Rigid Body*, which is a setting to determine the physical behaviour of objects. Rigid bodies allow to chain objects together, so that an impulse is passed from one chain link to another, hence their usage for ropes. Unfortunatly, we could not find a way to fix a rope on both sides and pull it into a curve. Furthermore, their is no easy way to adapt the number of chain links through python.



(a) A model with bones.

(b) The mesh comprised of 1024 vertices.

(c) Stretching is possible in Blender, but not in the BGE.

Figure 6.6: Scaling a bone in a certain direction does not work in BGE.

The two remaining ways we can think of drawing transitions is through shaders of OpenGL 2.0, as documented for Blender here [Kra15] or to hook a mesh through a hook constrain. But we admit here, that we simple lost faith in solving the task in BGE,

simply because our progress is rather sparse and the samples from which we could learn are non-existant.

6.4 Unsolved problems - Final analysis

We decided to give up trying to implement the stateroom editor as Blender game for a various number of reasons:

- Drawing a transition was not possible in a reasonable amount of time.
- Stretching an ellipse in a way that its text does not overlap can only be done in a cumbersome way through mesh manipulation. We discuss and solve this problem for OpenGL in 9.0.6.
- Centering the text inside the atomic state is an unsolved riddle.
- Calculating the anchor points for transitions, which must match pixel coordinates, is also unsolved.
- Where to calculate the routing of transitions is also an open question.
- There are no widgets like text input or combo boxes for the BGE. That means that an editor for source code, such as code snippets like *executable content* used in transitions, would have to be built from scratch.

Besides this, we have a lot of untouched features like: compound states, parallel states, aligning text with transitions, etc.

Building a stateroom editor as Blender game may be possible, but the development effort exceeds the resources of this work.

We want to note a key lesson we learned: Blender is at first a tool to model real world objects and to animate those models. To find concepts how things work in Blender or BGE, one must search after keywords someone might have used in their real world projects. For example, to visualize a transition, literal descriptions like, elastic line, bezier curve or a cylinder are valid. But these words are not objects in games, yet *ropes* and *chains* are. And those terms lead to the concepts one may looking for.

A final figure 6.7 shows the unfinished stateroom editor with some issues and some debug output.

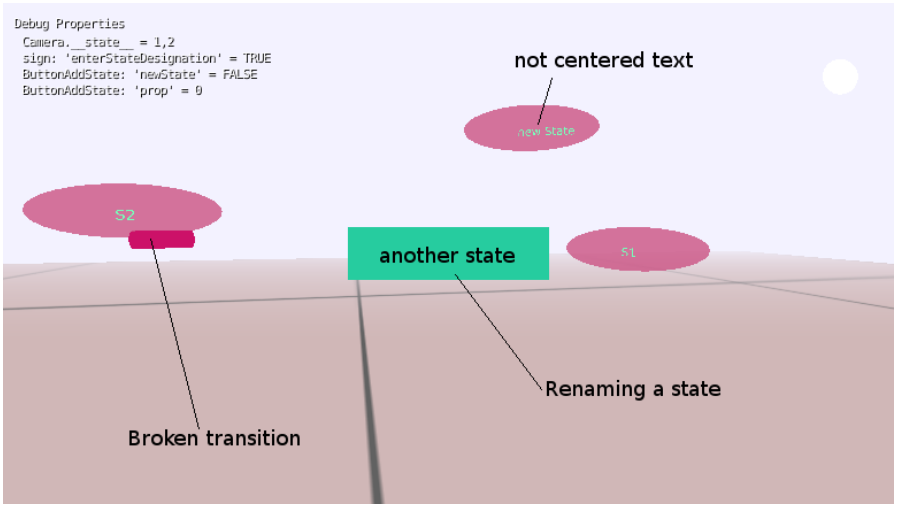


Figure 6.7: The stateroom with states and debug output. The transition cannot be controlled in BGE.

Part IV

OpenGL

Chapter 7

Introduction to OpenGL

OpenGL, OGL or GL stands for "Open Graphics Library" and is an interface for graphics rendering specified by *The Khronos Group*[Khr15]. It provides an API for applications to gain low level access to video cards through their graphics driver.

The main purpose of GL is to convert 3-dimensional objects, usually collected in a scene, into a 2-dimensional image, which is finally presented on a screen[1]. This process is called *rendering*. OpenGL reflected the render process in its **rendering pipeline**. It builds the key component of OpenGL and its understanding is imperative to successfully work with OpenGL.

Furthermore, it is important to understand that graphic adapters are designed and optimized to facilitate the calculations for rendering in a highly parallel manner. OpenGL does not conceal this characteristic. This fact must be kept in mind when designing applications using OpenGL.

OpenGL's history spans over more than two decades. Its latest release is version 4.5. Because our hardware driver from Intel support only version 3.3, the discussion is limited to this particular version. Also, the rendering pipeline may be used to apply filters on an image or to a previously generated output. We focus here on the transformation of 3d objects to 2d images. Furthermore, while we need a solid understanding of the rendering pipeline, our main interest is a specific part of it, the geometry and fragment shader.

This chapter gives an overview of GL's rendering pipeline, while the next chapter 8 builds the knowledge necessary for chapter 10, where the API is used.

The presented material is mainly summarized from [SA10], [SWH13], [Shr+13], [Ope15].

[1] Since version 4.2 a unit as an extension solely for computation is provided.[Sel15]

7.1 Rendering pipeline overview

What we simply called *rendering* before, is a long way of concatenated calculations. Such calculations are partitioned in stages and compendiously called *rendering pipeline*. Some of these stages are optional while others are mandatory. A greatly simplified pipeline of GL version 3.x is depicted in figure 7.1. Yellow polygons mark input/output data. Input data are information to render a specific part of the image (output data). Blue boxes mark programmable stages called *shader stage*s. Each shader executes a little program also called *shader*. Programs are written in the OpenGL Shading Language (GLSL), a C-like language. OpenGL 3.x supports three different shader stages, each equipped with special capabilities. Graphics systems can execute a number of shaders per shader stage in parallel. The amount of parallel processing shaders is hardware dependent and usually ignored by the developer[2]. Green boxes represent *fixed function* stages. They are highly configurable and provide tasks like collecting output data, interpolation of colors, perspective division or filtering vertices not contributing to the final image for any reason.

Before discussing each block of the pipeline in depth, we give a small overview of their interaction.

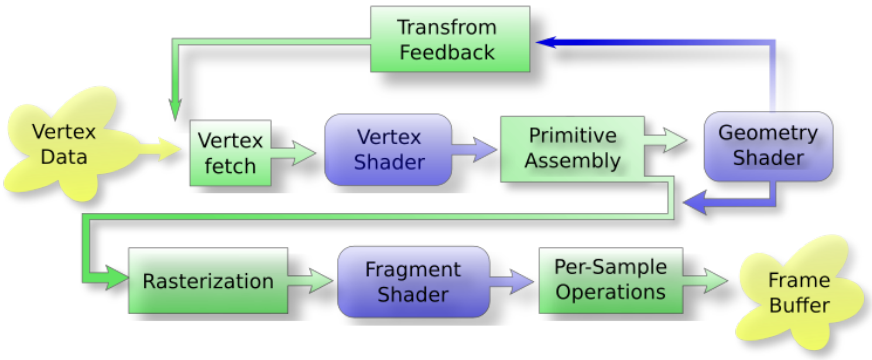


Figure 7.1: Rendering Pipeline for OpenGL 3.3. Input/output data is colored yellow, programmable units are marked blue and *fixed function* stages are shaded green. The Geometry Shader is optional.

The first yellow polygon, *Vertex data*, represents the input data, consisting of the scene's vectorial object descriptions and additional *attributes*. Each attribute is directly related to a vertex and contributes to determine its color and the gradient in-between neighbouring vertices. Examples for attributes are solid color values, material information or normals to reflect lightning. *Vertex data* is provided in a specific format, which is subject to the specification of *VertexAttributePointer*s, which is discussed in chapter 8.3.3. While this format describes which bytes belong to a data type, the data itself resides in GPU memory chunks called *buffers*. Such buffers are highly flexible as well as type-less. We discuss buffers in detail in chapter 8.3.

From the *Vertex Data* buffer, the *Vertex fetching* stage fetches a vertex with its attributes and feeds it to an instance of a **Vertex Shader** (VS). VSs operate on single vertices and

[2]The laptop we use is equipped with 16 shader units. Desktop computers have usually a couple of hundreds.

there associated data only. They have no information about neighbouring vertices. A common application for VS programs is to transform a vertex from one space to another, as discussed in chapter 3.1.

The outgoing stream of the VS stage is passed to the *Primitive Assembly* (PA) tier. In here, vertices are grouped into primitives. Such primitives are the basic geometric objects OpenGL can draw, like lines or triangles, and are further explored in section 7.2.4. There is no further investigation in PA in general, but the choice of primitive influences transform-feedback as discussed in 8.6.3. After grouping, the primitive is either passed to the optional **Geometry shader** (GS) stage, or it is immediately forwarded to the **Rasterizer**.

Assuming a GS is present, then each instance of it consumes a primitive from the PA stage. The strength of the GS is that it can change the primitive's geometry by discarding or adding vertices. Furthermore, it can be programmed to have read access to adjacent vertices, allowing to influence the internal calculations. This makes the GS ideal for interpolating arcs from two given endpoints and additional control points. More details about VS and GS are described in chapter 8.5.

The output of the GS stage is passed as primitive to the *Rasterizer*.[3] In this stage, each primitive is tested if it lies inside the viewport entirely. On a positive outcome, the primitive is kept for further processing, otherwise it is clipped or culled. Afterwards, the vectorial description is interpolated and broken down into discrete *fragments*. This happens inside the hardware and is mostly out of the developer's control. Fragments and pixels have a many to one relation, so, a pixel may be described by several fragments, while fragments belong to one pixel only. The number of fragments generated depends on the resolution of the viewport as well as on the objects projected onto it.

The generated fragments are passed to the **Fragment shader** (FS), while each fragment is equipped with information on how to color a single pixel. FSs determine the color of a single fragment without access to adjoin fragments. Common applications for FS are calculating gradients, mapping color from bitmaps to fragments or antialiasing. In section 8.5 we get to the bottom of shaders.

The last stage, *Per-Sample Operations*, consists of several steps to combine all fragments belonging to a coordinate to a final color value, displayed as pixel. Discussing all steps of this stage is beyond the scope of this work, but we elaborate some steps we experimented with in section 8.6.

At the end of a pipeline-run are the results stored into buffers and eventually presented in a window on screen. These buffers are managed by *framebuffer*s, which can be very complex. In section 7.2.2 we investigate into framebuffers; but just enough to render into several buffers at once.

[3] The *rasterization*-step is not uniformly defined across the references utilized. For example, fragment shader stage is described as part of rasterization in [SA10], while other sources describe them apart from each other, like [Shr+13, p.10]. The presented text emphasize the elements controlled in this work and follows rather the latter approach.

7.2 Prerequisites

Under Linux, OpenGL is a very powerful and flexible library. Its configuration has a huge number of options, leading to an enormous amount of combinations. We are only able to explore a fraction of it's possibilities. To not become lost in the standard we simplify some concepts that are vital to OpenGL-applications, but their details are rather insignificant in this work. That is mainly because they were initially set up, but touched sparsely during the rest of the project. Our main focus lies on the geometry and fragment shaders, because these two are the main shaders we use to produce and color the objects we want to render.

7.2.1 Binding concept

OpenGL manages internally a huge amount of objects to control the rendering pipeline. Most of these objects are referred to by a handle, denominated **name** in OGL terminology. An object can be manipulated by *binding* it's handle to a specific binding point and subsequently invoking functions that operate on the object identified through the binding point. Functions for manipulation expect either a binding point as argument or they have an implicit binding point; making it necessary to bind the object on the proper binding point. Binding points are also referred to as *binding target*.

While binding is an important concept of OpenGL and it is mentioned throughout this text, there is no need to engage in a deeper discussion.

7.2.2 Framebuffer

A framebuffer is a container, storing images where OpenGL renders into or reads from. Images are either *textures*, which are discussed in section 7.2.3, or *renderbuffers*[4], which we ignore in this document. For now, an image can be considered being a memory chunk, containing some image data. Before an image can be targeted by the render process, it must be associated with one of the framebuffer's *attachment point*s (AP)s, which are also denominated *buffers*.[5] OpenGL offers two types of framebuffers.

There is the *default framebuffer* (DFB). It is provided and controlled by the windowing system. Its buffers reflect usually the underlying hardware in its current state, like color depth, number of buffers[6], etc. A common DFB supports a front left buffer, which is visualized on the computer screen, a back left buffer, to render into and swap then with the front buffer, a stencil as well as depth buffer, which are discussed later in section 8.6.1 and 8.5.3. Front and back buffer are generally denominated *color buffer*s.

[4]Renderbuffers are memory chunks optimized for antialiasing (MSAA).

[5]Buffers, in this context are rather references to objects containing memory as representing memory by itself.

[6]Hardware supporting stereoscopic visualization need buffers for two sides (left and right).

Because the DFB's buffers are not replaceable by the application, they are rarely referred to as APs, but rather *buffers*.

The other type of framebuffer is called *framebuffer object* (FBO). FBOs are created and managed by the application and support features beyond the underlying hardware. They are not equipped with color buffers like the DFB, but support several attachment points[7] for color buffers. Into each attached buffer, can separately be drawn to, at the same pipeline-pass. To visualize a content of a color buffer, it is copied into the DFB, obeying its constraints. Rendering into several color buffers at once is known as *Multiple Render Targets* (MRT) and briefly discussed in 8.5.3.

7.2.3 Textures

A texture is a container for images; with an image being a one-, two- or three-dimensional array of image elements called *texel*s.Such containers manage several parameters to determine, the texture's *internal storage structure*, the *format* of the stored texels and the way texels are *sampled* upon usage.

Before analysing the configuration used in this project, we contemplate what kind of graphics we want to display. We use two types of graphics, icons and glyphs. Icons, for example, are used for menus and are discussed in chapter 9. They are drawn in RGBA color format. Glyphs are used to display text and discussed in chapter 10.5.2. They are retrieved in gray scale, but converted in the RGBA format. To facilitate the discussion, glyphs and icons are referred to as icons in this chapter. OpenGL can only manage a certain number of textures, in the magnitude of dozens. That makes it cumbersome to have one texture per icon, considering that we have around 100 icons; 2 * 26 letters, numbers, special character, icons, etc.

So, the common way to organize a larger number of icons, is to use a technique known as *texture atlas* (TA), or *texture map*. A TA is a huge image, containing all icons to display. For each entry in the TA, a coordinate and size is stored in a look-up table. OpenGL supports TAs by accessing a single icon through its coordinate and size parameters from the look-up table. These coordinates, and therefore also the size, are usually measured in unified UV coordinates. That means the size of the texture is 1 in x (called U) and 1 in y (called V) direction. The look-up table is the responsibility of the developer and not a component of OpenGL.

To display a texture, it is mapped to vertices and passed to the *Primitive Assembly* stage. Here the texture is *sampled*, which we discuss shortly, to fit the area covered in-between the mapped vertices. For example, assuming a chessboard-texture is mapped with the UV coordinates (0.0,0.0) to (0.75,0.75) to 4 vertices, which are placed as square on the screen. Because the texture is addressed to $\frac{3}{4}$ in each direction only 6 rows and 6 columns are drawn.

[7] OpenGL 3.3 specifies at least 8 attachment points.

After this short introduction into textures, we present and argue for the configuration used.

The storage structure specifies, among other things, the dimension of a texture. Since we have only a 2-dimensional texture, we choose **GL_TEXTURE_RECTANGLE**. It is the only storage structure for 2D only textures. Furthermore, it allows to access the texture in pixel coordinates, rather than UV coordinates; saving the calculations of the uniform coordinates.

The color format was already mentioned, RGBA (GL_RGBA). We give the alpha channel a special meaning for images (not glyphs). It is used to detect easily which pixel has a color, and which has none. This is discussed in chapter 10.4.1 to prevent invisible pixel from becoming clickable with the mouse pointer.

The last parameters to configure determine the way a texture is sampled. While UV coordinates operate in the range $[0, 1]$, the texture can be accessed beyond this range. The settings GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T determine how texels are retrieved in U, respective V, direction, when a texel's coordinate does not lie in UV's range. We do not access the texture outside its boundaries, which makes this settings irrelevant to us. But since it is mandatory, we clamp the coordinates to retrieve to the range $[0, 1]$ with the option GL_CLAMP_TO_EDGE. Another parameter configures how the texture is magnified, respective minified, in case the area to which the texture is to be mapped is larger, respective smaller, than the texture itself. The responsible options, GL_TEXTURE_MIN_FILTER and GL_TEXTURE_MAG_FILTER are both set to GL_LINEAR, which determines the color for a pixel by performing a weighted linear blend between neighbouring texels.

With this configuration, the texture, from the OpenGL point of view is basically set up. We spare to list the function to create and manage the texture.

To complete the texture atlas, a second texture is used to store the size and the location of the icons kept in the previous set up texture. The texture *storage* used is **GL_TEXTURE_BUFFER**. This kind of buffer does not apply interpolation, nor is it accessed via uniform coordinates, hence, there is no further configuration. It is just a texture providing **random** access to an underlying buffer. We discuss this further in section 8.3.2 together with the underlying buffer.

The sample parameters, which we used earlier, are actually used to configure a default sampler, automatically provided with the texture. But textures can be accessed through additional *sampler objects*, which are out of scope of this work. Nevertheless, in section 8.5.2 we discuss how default samplers are accessed inside a shader program.

7.2.4 Primitives

Primitives are the entities a graphics adapter can draw. All objects to render, like cars or trees, must be constructed out of these basic entities. OpenGL offers the primitives illustrated in figure 7.2. **Points** are single coordinates and can have a size up to 40 x 40 pixels. **Lines** connect two coordinates with a maximum width of 1 pixel. A *line strip* and *line loop* exists to draw line strips and loops more efficient by saving memory for coordinates. The **triangle**-types are used to form larger areas.

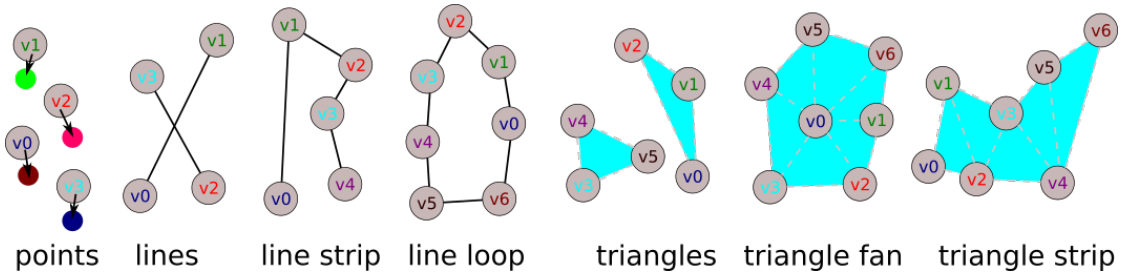


Figure 7.2: The Primitives OpenGL can draw. Dotted, light gray lines mark the triangles and are not visible when rendered.

What kind of primitive to draw, must be selected upon rendering start. The choice of primitive does not just affect the resulting geometry and clipping, also the amount of vertices a GS pulls is determined. This is elaborated further in section 8.5.2.

A single render-pass can produce thousands of primitives, but all of the same type. Usually, a scene is constructed out of different types of primitives. That makes it necessary to run several render-passes. This procedure, transforming a scene into an image, is discussed in section 8.2 further.

We argue throughout chapter 10 for the choices of primitives we use to construct our objects.

7.3 Draw call

The rendering process is started by issuing a *draw call*. There are different kinds of draw calls, influencing the pattern vertices are pulled. We go briefly through the ones we implemented during the project.

All presented draw calls have a *mode* parameter. This parameter determines which primitive, from the ones in figure 7.2 on page 66, is grouped together by the primitive assembly stage and subsequently rendered by the rasterizer.[8] Table 7.1 shows in addition to the modes one could expect for primitives, some modes with the ending ADJACENCY. Adjacency modes give read access to neighbouring vertices inside a geometry shader. We do not discuss the usage of *adjacency*-modes further.

We mentioned earlier that vertices, and their attributes, are provided by buffers. When discussing draw calls, vertices and attributes are collectively called elements.

[8] A geometry shader can overwrite the mode as we discuss in section 8.5.2.

GL_POINTS		
GL_LINE_STRIP	GL_LINE_LOOP	GL_LINES
GL_TRIANGLE_STRIP	GL_TRIANGLE_FAN	GL_TRIANGLES
GL_LINES_ADJACENCY	GL_LINE_STRIP_ADJACENCY	
GL_TRIANGLES_ADJACENCY	GL_TRIANGLE_STRIP_ADJACENCY	

Table 7.1: These draw call modes render the primitives introduced in section 7.2.

- **glDrawArrays** is the most basic draw function. Its arguments are the number of elements to render and an offset to the buffer's first element. This is the only draw call we finally use.
- **glMultiDrawArray** is equivalent to multiple calls of glDrawArrays. While each primitive is from the same type, it may consists of a different amount of elements.

  This draw call is used in chapter 10.4.2 to draw several polygons at ones.
- **glDrawArraysInstanced** is equivalent to multiple calls of glDrawArrays. This call can be used to draw multiple instances of the same elements. The identifier **gl_InstanceID** is available at the VS stage and gives the opportunity to change the elements belonging to an iteration. It can be used, for example, as index to access an array of positions.

  But there is another option to pass individual data per iteration. The **glVertexAttribDivisor** allows to configure a buffer in such a way, that an element is only pulled when n elements were pulled before.

  We use this draw call in chapter 10.4.1 to draw ellipses.

Chapter 8

Rendering pipeline

The rendering pipeline transforms vectorial descriptions into discrete color values, shown on the screen.

This chapter explores the possibilities on how vectorial descriptions can be fed to the graphics adapter and how the pipeline can be programmed to transform these descriptions into color values. We talk in particular about the organization of buffers, the programming of shaders and the preprocessing of samples at the end of the pipeline. Before OpenGL can commence its work in general, it must be initialized and the default framebuffer configured. We omit a discussion about initialization and configuration since they are more or less standard procedure and are puny for the actual editor. With one exception, the viewport setting.

While we are interested in shader programming, in particular in the geometry and fragment shader, a deep analysis of the GLSL, the language shaders are programmed in, is out of scope of this text.

8.1 Viewport setting

The viewport has a fixed position in OpenGL but its size must be set prior rendering. This is done with the function **glViewport**. It specifies width and height of the viewport in number of pixels, usually set to the size of the window. It's center has the global position $(0, 0, 0)$ while its X/Y axes are aligned with the global ones. GL uses the left-handed coordinate system, known from chapter 3.1. From that follows, that the user looks into the $-Z$ direction. Two further arguments move the viewport to an arbitrary position on the projection plane. We determine the viewport setting in chapter 10.2.1.

8.2 General Process

Assuming OpenGL is setup, the pipeline can be prepared for rendering. Since only a single primitive type can be drawn at once, several pipeline passes are usually necessary to produce a frame.

The principle drawing cycling is:

- Update global information used by the shaders (projection matrix, camera position, etc.)
- Clear the target framebuffer.
- To draw a frame:
    1. Fill and attach buffers to pipeline
    2. Compile and upload shaders
    3. Issue draw call
    4. Repeat until all primitives are drawn
- Switch framebuffer

The rest of this chapter is devoted to the OpenGL components, used in chapter 10 on page 89.

8.3 Buffer Objects

Buffer Objects (BO)s are containers to manage raw memory blocks. These memory blocks store data like vertex positions, colors, matrices or anything else one can think of to calculate the objects drawn on the screen. They are usually filled from the host side and its content is drawn on the screen. But buffers can also capture the output of the transform feedback stage and pass it back as input to the pipeline. We focus on using BOs for the first purpose and discuss the latter in section 8.6.3.

OpenGL offers a couple of different kinds of BOs. They discern in their maximum storage size, the way as well as location they can be accessed and other parameters, which are of less interest to us. Before we analyse the details, we elaborate first how buffers are generated and modified in general.

Buffer objects are administered by GL and represented by *name*s. Such a *name* can be retrieved by calling **glGenBuffers** which reserves storage for BO's internal structure, but none for the actual buffer. To allocate memory for the buffer the BO must first be bound with *glBindBuffer* as described earlier in section 7.2.1. Possible BO's targets and their general purpose are listed in 8.1. A chunk of memory can then be requested by invoking **glBufferData** with the same target the buffer object has been bound before.

Target name	Purpose
ARRAY_BUFFER	Vertex attributes
COPY_READ_BUFFER	Buffer copy source
COPY_WRITE_BUFFER	Buffer copy destination
ELEMENT_ARRAY_BUFFER	Vertex array indices
PIXEL_PACK_BUFFER	Pixel read target
PIXEL_UNPACK_BUFFER	Texture data source
TEXTURE_BUFFER	Texture data buffer
TRANSFORM_FEEDBACK_BUFFER	Transform feedback buffer
UNIFORM_BUFFER	Uniform block storage

Table 8.1: Buffer binding points for GL 3.3.[SA10, p. 39] The purpose is mostly a guideline.

To give OpenGL a chance to optimize buffer usage, **glBufferData** takes a hint passed as argument. We use DYNAMIC_DRAW, which is the right choice if its content is modified regularly by the application and it is used to draw by OpenGL. A deeper discussion is out of scope of this document since this is a rather a topic for applications squeezing out the last bit of performance.

Initial content for the buffer can be passed to **glBufferData** via pointer, pointing to a chunk of host memory. Its size must be identical to the one requested on the GPU side. Afterwards, the content can partially be overwritten by **glBufferSubData**. As for glBufferData, the buffer's BO has to be bound at first.

A buffer is freed when its BO is destroyed or a new buffer is assigned to it.

We look now into the characteristics and applications of buffers.

8.3.1 Uniform buffer object

OpenGL offers for constant data, which do not change during draw calls and have the size of a few kilo bytes only, a special method of access. Any variable, or structure can be used for that kind of access by using the *uniform* qualifier; converting them to *uniform variables*. These varialbles are than integrated into the shader program, which prevents them from being shared with other shader programs.

Uniform buffer objects (UBO)s are used to collect and handle several *uniform variables* in a single object. This allows not only to share the collection among shader programs, it also offers a comfortable way to pass several uniform variables to several shader programs. Furthermore, the upload of the same data several times is prevented. Routing the data through the application, so that it can be uploaded, is also prevented because, a UBO is registered to the shader once upon initialization only, while uniform shaders have to be uploaded per draw call.

UBOs come with a size limitation which is 64KiB on our system.

We use this kind of buffer in section 10.5.6 to share uniforms among shaders.

8.3.2 Buffer Texture

Texture buffers (TB) are read only texel arrays which can be randomly accessed from inside a shader. A lookup of its content is a achieved by calling *texelFetch(gsamplerBuffer sampler, int index)*. The first argument is a sample buffer. That is because TBs are treated internally as textures, albeit they do not perform interpolation like general samplers. The second argument is the index of the desired texel.

OpenGL makes TBs minimal size at least 64Ki texels, whil our system support 128Mi texels.

We use buffer textures to lookup dimensions about glyphs in section 10.5.2 and store information about SCXML states in section 10.4.1.

8.3.3 Vertex buffer object

Vertex buffer objects (VBO)s are buffers of arbitrary size to store vertices and their attributes, like coordinates, colors, normals, etc. Contra the mathematical discription, where a vertex **is** a coordinate, for vertex buffers, a vertex is an empty entity and its coordinate is a separate attribute. VBO's are the source for the *vertex fetch* stage which pulls the data consecutively, passing it to the shader instances. The amount of bytes each shader receives is determined by the *Vertex Attribute Pointer* which we discussed in a jiffy.

Because buffers are not resizable, it is necessary to either know their maximum size in advance or to keep track of it and reassign a bigger chunk of memory as soon as needed. Our application, and with it the buffer size for the objects to draw, depends on the input of the user. That makes it impossible to predict the size of the buffer. Furthermore, VBOs are used quiet frequently since they are the source of basically each object on the screen, that together makes is practical to encapsulate their management into a class.

There is no explicit size limitation for VBO, except implicit limitations, like size of host memory.

Vertex Attribute Pointer

Assuming a buffer is filled with an array of a struct, as shown in listing 8.1, containing information like color and id. Than GL needs the struct's layout to read each variable from its proper memory location. Furthermore the mapping from the struct's entries to the vertex shader input variables, called *vertex attributes*, must be given to GL. An example for vertex attributes is listed in 8.2 and will be elaborated next chapter about shaders.

```
struct {
  int   id[2];
  float color[4];
} array[100];
```

Listing 8.1: An array of vertex attributes.

```
in ivec2  id;
in  vec4  color;
```

Listing 8.2: Two vertex attributes as VS preamble.

Vertex Specification is the process of mapping the struct's members to the shader's vertex attributes by using the buffer's layout. This mapping occurs inside the *Vertex Attribute Pointer* (VAP) object. When creating a VAP, the buffer to pull from must be bound to the target GL_ARRAY_BUFFER. In figure 8.1 a schematic shows how the VAP connects buffers and vertex shader instances. The **buffer id** is copied from the bound buffer when the VAP is defined. So, when the VS pulls this specific attribute, it knows which buffer is to address. The **type** and the **number of components** form the GLSL variable type. For the first attribute, 4 floats are taken from the buffer and appear as a *vec4* in the VS. The **stride** parameter is the size of a struct or zero for tightly packed, non-interleaving data. For interleaving data, an **offset** specifies the location inside the struct of the attribute. To couple the VAP with a shader atrribute (input variable), the attribute location is first queried from the shader. This location is then placed as **location** argument when the VAP is defined.

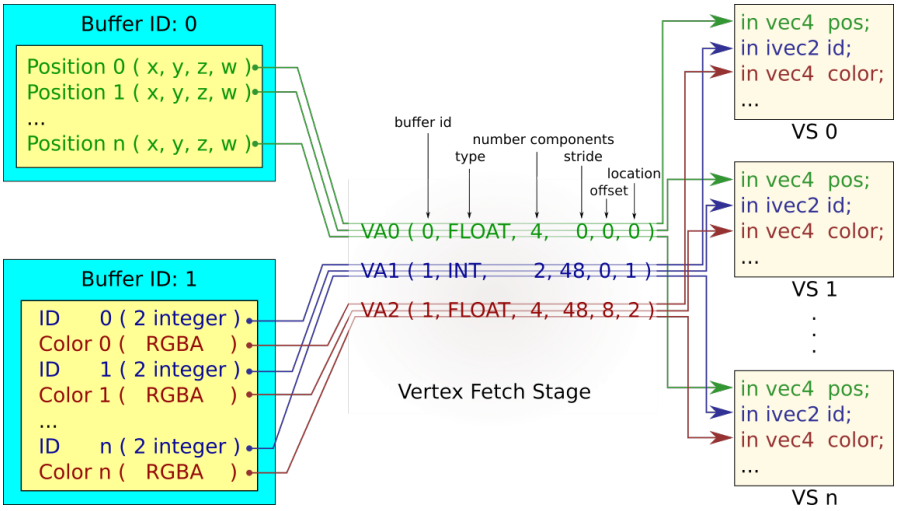


Figure 8.1: Fetching vertices from different buffers according to vertex attribute pointers.

VBOs and vertex attribute pointer are managed by the library helper_gl, presented in chapter 10.5.3, and used throughout chapter 10.

8.4 Vertex Array Object

A Vertex Array Object (VAO) is a wrapper objects, storing data about VBOs. VBOs are usually setup once and then used throughout the life-time of the application. Changing the contents of the buffers connected to VBOs does not affect the VBOs itself. But VBOs are connected to entry slots to feed shader programms with data. These entry slots have a hardware limitation. Instead of assigning every time VBOs to the shader program from the start, they can be collected in a set, which is controlled by a VAO. Everytime a different shader program is loaded, only the VAO, for that shader program, has to be bound. That disables previously bound VAOs automatically and only VAPs assigned to the VAO are present. We do not pursue VAOs further.

8.5 Shaders

Shaders perform application-defined calculations on the GPU. They transform data provided through the aforementioned buffers by applying algorithms programed as **shader program**. Such a shader program consists of programs from a vertex shader (VS), a fragment shader (FS) and optional, a geometry shader (GS). While each shader is programmed separately, they are compiled and linked together before transfered as single binary blob to the GPU. The term *shader* is often used ambiguous for shader program, shader stage or instance of a shader.

Before we look into the shaders we elucidate how the data channels between shaders are *piped* together.

Each shader stage offers two sets (input/output) of variables. These variables are refered to as *varyings*; constructing logical pipes between shaders, hence the term *rendering pipeline*.[1] There are built-in varyings as well as user-defined varyings. Built-in varyings depend on the shader stage and are explaint, if used, in the shader sections below. User-defined varyings must be defined as pairs. An input variable on one shader stage is expected to have a matching output variable on the previous shader stage. The qualifiers **in** and **out** determine the belonging set. A typical preamble with varyings and qualifiers of a vertex shader is presented in listing 8.3.

```
     in   vec2  positionStream; // A float array of 2 elements
     in   ivec2 letterStream;   // An integer array of 2 elements

     out ivec4 letters;         // An integer array of 4 elements
flat out int   slotPosition;    // A single integer
```

Listing 8.3: User-defined variables in a shader. The **in** qualifier declares incoming variables, while the **out** qualifier marks variables passing data to the next stage.

To simplify the further discussion we assume that no GS is present so that the VS is linked to the FS, with the primitive assembly (PA) stage and the rasterizer sitting inbetween; as illustrated in figure 7.1 on page 61.

It follows some characteristics of each shader stage.

8.5.1 Vertex Shader

The vertex shader is the first fully programable stage. As discussed earlier, it receives its data, determined through **Vertex Attribute**s, from the *vertex fetch* stage. For each vertex a separate instance of the vertex shader program is initiated. VSs are ideal for plain vertex operations where neighbouring vertices can be ignored. Each incomming vertex produces exactly one output vertex. That means, neither additional vertices can be generated nor can incoming vertices be dicarded.

[1] The term *varyings* is not to be confused with the storage qualifier varyings, which is obsolete. See [KBR10, pg. 27] and [SA10, pg. 75].

We use vertex shaders merely to pack/unpack attributes and to forward them to the geometry shader.

8.5.2 Geometry Shader

Unlike a VS, a GS can consume up to three vertices and can in read in addition three adjacent vertices. We do not consider adjacent vertice further. The input primitive type determines in the preamble of the GS, line 1 in listing 8.4, how many vertices are pulled from the vertex buffer. This type must correspond to the *mode* specified by the draw call, according to table 8.2. The vertices and their varyings are then accessible through arrays, defined is counter part of the VS's output varyings.

The GS has also an output primitive type, which is either *points*, *line_strip* or *triangle_strip*; 2nd line in listing 8.4. Changing the output primitive overwrites the initially, through the draw call declared primitive type.[2] That allows to draw, for example, triangles by just specifying their position by single vertices.

```
layout (points) in;                                  // input primitive type
layout (triangle_strip, max_vertices = 16) out;      // output primitive type
```

Listing 8.4: Input/Output declarations for a geometry shader. Input declaration must match the draw mode.

A GS instance can produce more than one primitive, but not more vertices than specified through *max_vertices* (2nd line). Input, as well as output primitive types cannot be changed during pipeline processing. Our library *text_3d* takes advantage of these two features, as described in chapter 10.5.2.

Incoming data can be transformed by the GS with algorithms written in GLSL.

Outgoing data has to be grouped in respect to a vertex. This vertex has to be submitted through the built-in varying **gl_Position**. User-defined varyings may follow or be prepended before the vertex is submitted. A group is finalized by calling **EmitVertex**, signaling the hardware that a vertex and its descriptional data is ready.

Since a GS instance can issue serveral primitives, a similar command is used to locked a primitive, **EndPrimitive**.

Next section emphasize how these data is pre-processed for the FS.

GS input primitive	draw call mode	vertices pulled
points	POINTS	1
lines	LINES, LINE_STRIP, LINE_LIST	2
lines_adjacency	LINES_ADJACENCY, LINE_STRIP_ADJACENCY	2 (2)
triangles	TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN	3
triangles_adjacency	TRIANGLES_ADJACENCY, TRIANGLE_STRIP_ADJACENCY	3 (3)

Table 8.2: The draw call mode must match the GS input mode, but GS can output in a different mode. Unless a TF is activated.

[2] When transform feedback is activated, this freedom is lost.

We use GS's to apply projections and calculate objects positions. Furthermore, are they used to locate specific regions inside a texture. See chapter 10.2.3 as example.

8.5.3 Fragment Shader

Varyings send through the pipes to the VS are rasterized and segmented into fragments. The choice of primitive, given through the draw call or changed by the GS, determines how the rasterizer interpolates between the vertices and their accompanying varyings. Furthermore, the built-in varialbe **gl_Position** takes a homogenous coordinate, allowing the rasterizer to perform the perspective division. This is important, because the perspective divions also influences the interpolation of the varyings. The interpolation can be controlled with the *interpolation qualifier*s: **flat** - no interpolation, **noperspective** - linear interpolation and **smooth** - interpolates while taking perspectivity into account.

The resulting data are then passed as *fragments* to the FS.

Any kind of algorithm can then be programed in GLSL to finally determine the color of the fragment. But we are especially interested in orienting ourselfs inside the FS, so that we can determine its color, depending on its position inside the object it belongs to.

Assuming a screen aligned quad with a texture on it. Then the GS is programed to use *triangle_strip* and issues 4 vertices. Because the vertices, and therefore the resulting fragments, can be everywhere on the screen, it is impossible for a fragment to determine where it is located on the quad. If we issue for each vertex of the quad a specific coordinate, the interpolation of the rasterizer gives each fragment a unique varying. For example, each vertex becomes one of the following 2D coordinate as attribute: {(1,1), (1,0), (0,0), (0,1)}. The fragment in the middle of the quad, has then a location attribute with the value (0.5, 0.5).

We exploit this behavour numerous of times. Worth noting is chapter 10.2.3, where the color depends on the mouse pointer's position, and chapter 10.4.1, where the geometry of the atomic state is adjusted depending on its size.

Multiple Render Targets - MRT

Multiple Render Target (MRT) allows to bind several color buffers to a framebuffer object. The fragment shader can then write different colors into each buffer. A drawback of this operation is that it cannot be written into the default buffer. Hence, if one of the buffer shall be visible on the screen it must be copied into the default framebuffer. But this is usually a fast operation.

We use the MRT in chapter 10.2.2 to identify objects under the mouse pointer.

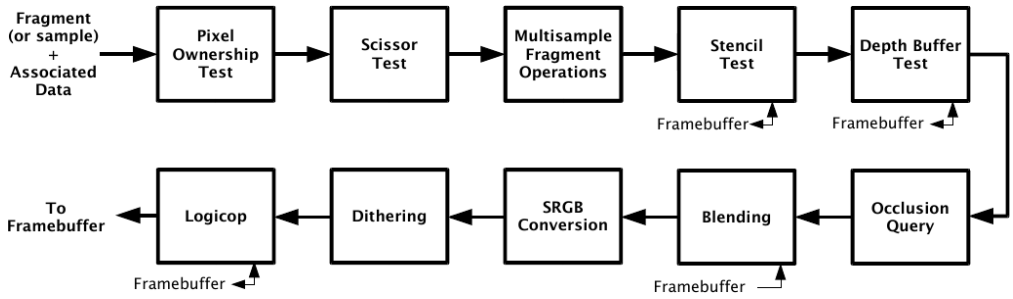


Figure 8.2: This work benefits only from the Depth buffer and *Blending* stage[SA10, p. 195].

8.6 Per-Sample Operations

After the fragment shader finishes execution, a series of fixed function operations combine the fragments into a final pixel color. Figure 8.2 shows the path of this final stage. While most of these functions are disabled by default the remaining do not need to be configured to work probably. We take a closer look at *Depth test* and *Blending* only.

8.6.1 Depth Test

The depth test discards fragments depending on their distance, a test function and a value stored in the depth buffer. An incomming fragment's distance is tested against the distance stored in the depth buffer. That means the depth buffer has the same geometry as the near plane. If the test fails, the fragment is discared and not processed further. In case of a positive outcome, the fragment passes and is processed by the subsequent function blocks. Furtheremore, the depth buffer is updated with the distance from the fragment just processed. How the test is performed, can be determined with the **glDepthFunc**. Possible comparing operations are: NEVER, ALWAYS, LESS, LEQUAL ($<=$), EQUAL, GREATER, GEQUAL and NOTEQUAL.

When a new frame is rendered, the depth buffer should be cleared by setting a default value through the function **glClearDepth**. The default value depends on the function used for comparison.

8.6.2 Blending

Blending combines the color of incoming fragments with either the color already stored in the framebuffer or a constant value. Afterwards the resulting color is stores back into the framebuffer. The general idea can be expressed as equation:

$$C = EQS(fragmentcolor) + EQD(framebuffercolor)$$

. While EQS as source equation and EQD as destination equation and C as the color stored back into the buffer. OpenGL offers a huge amount of equations to control the

influence of each of the color channels. Our goal are transparent objects, so that a SCXML state does not necessarily occlude another widget. According to OpenGL[SG06] this is best achieved by choosing for $EQS\ GL\_SRC\_ALPHA$ and for the destination $GL\_ONE\_MINUS\_SRC\_ALPHA$). Further discussion about these equations is beyond the scope of this text.

The above blending equation is not commutative. As a consequence, changing the drawing order of superimposing objects ends in different color values, hence appear different on the screen. To avoid this affect the drawn objects can be sorted in z direction, starting with the farthest object away, cp. [AMHH08, p. 136].

8.6.3 Transform feedback

Transform feedback[MF12] TF as a technique to store the output of a VS or GS in a buffer and feed it back as input in a subsequent draw call. We experimented with TF for all kind of calculations but finally abandoned its usage because it does not support the usage of user-defined structures; data has to be provided in textures. That makes it hard when restructuring the code over and over again.

One interesting observation is that when using TF, the GS can not overwrite the output primitive, because the TF expects the output primitive given from the draw call.

Chapter 9

Design

In chapter 1 we reason for a new approach to design software by using state machines. We argued that the developing of state machines can be enhanced by laying out the statecharts in a 3-dimensional room, so that developers learn easier the logic behind the state machine.

Laying out states demands a new design and handling for the elements SCXML provides. But furthermore, menus and icons are needed to manipulate and steer the system.

This chapter is devoted to this topic.

9.0.1 Floating and stationary states

To facilitate the further discussion, we introduce the notion of *floating* and *stationary* states. Compound-states as well as parallel-states are considered as *floating* states. That is because these states enclose a volume, defined by their children as well as their SCXML-id. Nevertheless, the volume is rendered as a 2-dimensional polygon, as we define in chapter 9.0.7. Atomic-states and pseudo-states are collectively named *stationary* states. Their size is fixated in global space and does change when the point of view does. While they are rendered 2-dimensional too, they do not depend on underlying states. We discuss these states in chapter 9.0.6.

9.0.2 Anchors

Anchors are specific locations where two objects touch each other. Assuming a transitions, for example, connecting a source state with a destination states. From a graphical point of view, the transitions has two anchor points, one on each state's hull. We name these anchors **graphical anchors (GA)**, describing a specific position on a graphical representation. Occasionally, we want to refer, event when discussing

graphical matters, to a relation between transition and a state where the GA is of no concern. In that case, we speak from **logical anchors (LA)**, refering to the fact that a transition is connected to a state. By using the notion LA, we free our minds from the idea if the transition is embarking from a state or if it is targeting the state.

9.0.3 Label

several attributes:
id: <state>, <parallel>, <event>, ..
type: <send>, <invoke>, ..
name: <param>, ..
expr: <param>, ..
location: <param>, ..
target: <transition>, ..
event: <transition>, ..

A very elementary item is the **label**. It shows one short line of text, mostly a single word. The intended use is to display user-changable SCXML-attributes like *id* of states or *event*s of transitions. The label itself is not clickable and has no background color. It's position is updated, either by the layout algorithm or in conjunction with other objects. For example, when the user relocates a state or the camera is moved.

Labels are mainly implemented through the text_3d library explained in section 10.5.2. That is because the library displays text in 3D space, yet screen aligned to ensure legibility, as we argue in chapter 3.3. Because text_3d can only handle single lined text, executable content, which expands usually over several lines, is not visualized through the label. Widgets containing executable content hark back to Qt's internal widgets.

9.0.4 Menu

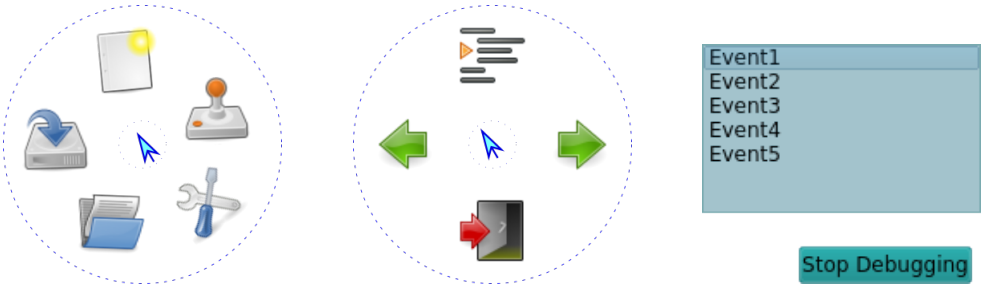
several attributes
Context menu: depends on underlying object.
General menu: none.

Like most GUIs, we provide a menu to give the user easy access to actions. It can be partitioned into three categories or modes. The first category, the *general menu*, handles the options: Load, Save, New, Preferences and Simulator, see figure 9.1a. Load, Save and New are file related and self explanatory. The Preference option opens a dialog to adjust some settings, for example movement speed or the FOV. The simulator option activates the debugger menu. The *debugger menu*, shown in figure 9.1b allows to interprete a state machine and contains the options: ShowEventList, Start (arrow right), Exit and Stop (arrow left). With the option *ShowEventList* the the window in figure 9.1c can be toggled. It lists such events, which can be taken from the current configuration. Double clicking an entry sends the event to the state machine. The third category, the *context menu*, offers actions to manipulate the underlying object, like adding a transition or changing an attribute. These menus are elaborated on the objects they influence.

The rounded menus in figure 9.1[1] are known as pie menu [Hop91]. They ensure faster access than conventional menus by using less screen space.[Sam11]

Implementation details about the menu are elaborated in section 10.2.3.

[1] Icons are taken from the GNOME project (tango-icon-theme)[pro09]



(a) The *general menu* with the options: **New**, **Simulator**, **Preferences**, **Load**, **Save**.[1]

(b) The *debugger menu* with the options: **ShowEventList**, **Start**, **Exit**, **Stop**[1]

(c) During debugging, a list of such events, resulting, if occure, in a state change are offered for emission.

Figure 9.1: Menus are implemented as Pie chart menus. [Hop91]

9.0.5 Tooltip

Tooltips are shown for attributes which are likely not to be changed on a daily basis. Neither are the information they display helpful or necessary to understand the state machine's essence. For example, *version* and *name* are attributes of <*scxml*> and serve an administrative purpose. They do not influence the state machine. Nevertheless, these information are important and must be presented and edited on the user's demand.

A tooltip is shown temporarly when the mouse hovers over an object. When click upon, its content can be changed. The places where tooltips are used is mentioned in the sections below.

9.0.6 Stationary states

Final state

A final state signifies the end of a parallel's region or the end of the state machine. We searched the internet for the keywords *end icon* and *final icon* to see what kind of symbols are connected to such a state. Regarding to our findings we designed figure 9.2, a dart like icon. The three triangles reflect the presents (or non-present) of the three children tags. With <donedata> being the one directing to the bottom, <onentry> the one towards the middle and <onexit> the triangle directed to the right. A brighter filled drawn triangle (like donedata) indicates the present of content, while a darker holey triangle (remaining triangles) shows that a tag is not present.

<final>

tags:	attributes:
<onentry>	id
<onexit>	
<donedata>	

Figure 9.2: Final state icon.

Atomic state

<state>

as atomic

tags:	attributes:
<onentry>	id
<onexit>	
<datamodel>	
<invoke>	

The common representation in 2D editoris for states is a box, or a box with rounded corners. To build hierarchies, boxes are placed into each other. Figure 9.3 shows a parallel state with atomic states and an hierarchical state in a ecplipse plugin to edit state charts. Our approach in Blender, see section 6.3, exhibits a state as ellipse. There are three reasons to consider an ellipse rather than rectangle or rounded rectangles.

First, an ellipse can be expressed in a mathematical function. That way there are no edge cases, like rounded corners to consider when calculating GAs (graphical anchors). The second reason to draw ellipses rather than rectangles is the adaption of the GA during the movement of the user. This is different in 2D applications where the routing for transitions is done once and only adapted when the user edits the chart. Imagine a user moves and all GAs are adapted because of the billboarding. That makes it possible that a GA is recalculated and wanders around the hull. Assuming rectangles are used to represent states, then a GA could halt on an edge of a rectangle for some time. Such an abrupt upholding may irritate an observer, tricking him into thinking something went wrong. Considering the same movement of the GA around an ellipsoid, does not solve the problem entirely, because the object triggering the GA's recalculation has a larger circle than the ellipsoid, which means that several positions of the triggering GA are mapped to a single position on the recalculated GA. Nevertheless, with an ellipsoid the effect is diminishes. The last reason is that rounded objects look more pleasing to the human eye.

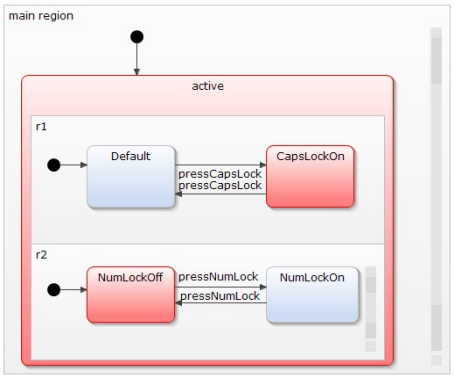


Figure 9.3: Yakindu, a state chart editor, draws states in rounded rectangles. [AG15]

Figure 9.4 shows the first mock-up. We tried in the initial design phase to make the state as expressive as possible and reduce objects which can break homogeneity. Transitions, which loops to the same state are such elements, since they are not drawn away from the state, but turn quickly and return to the source. The red circle indicates there is a transition loop without occupying the space around the state. Neverthelss, such an indicator has some downsides. First of all, it breaks the notation that transitions, no matter what kind, leave the state, which is a think what developers would simply expect. Secondly, there is no space for displaying the triggering event label. This could be indicated by adding a simple point in the middle of the loop-indicator, which is still not the same as having the event label displayed. Another point is that it might be easily overlooked, especially when the state is farther away. We do not pursue this notation.

Figure 9.4: First mock-up of a state. The circled arrow signifies a transition to itself.

In chapter 10.4.1, is the atomic state's evolution and its final presentation discussed.

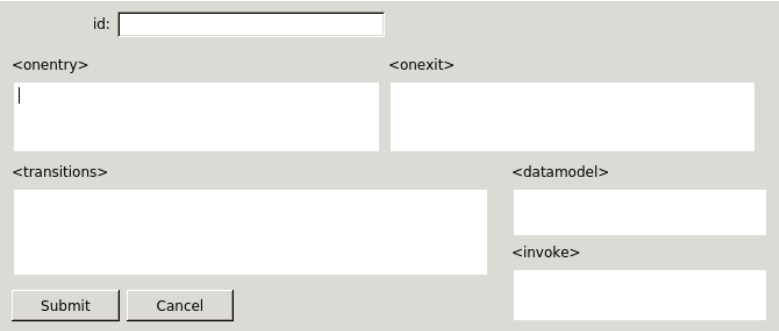


Figure 9.5: Dialog to edit source tags for atomic states as well as compound states.

Pseudo states

Initial state The initial state is drawn as a circle with an **I** in it, see figure 9.6 to the right. Every compound state has an initial state, as discussed in section 9.0.7. We do not differentiate between an <initial>-tag element and an initial-attribute of a <state>. Both are drawn the same way. There are no further options.

Implementation details are listed in section 10.4.1.

<initial>, initial=""

tags:	attributes:
<transition>	id

<history>

tags:	attributes:
<transition>	type, id

Figure 9.6: Icons for history state (l) and initial state (r).

History state History states are drawn as circle too, but with an **H** in it, as illustrated in figure 9.6 on the left side. While the default *type* (shallow) is not explicitly indicated, it can be changed through the tooltip facility to *deep*. This prints the word "deep" below the icon and ensures that the icon is distinguishable from other icons and from its default setting, even at a greater distance. We considered to negate the icon or to put an asterisk on its top right corner, like its original visualization in figure 2.4 on page 12. But showing the type is not just easier to differentiate, it also reminds the developer what this option actually means in contrast to an asterisk.

Implementation details are listed in section 10.4.1.

9.0.7 Floating states

Compound states

Displaying compound states in 3 dimenions is not completely solved yet. We discuss on the hand of the illustration in figure 9.8 the trouble of the subject. Figure 10.9 shows 3 overlapping compound state mockups, each drawn with transparent background in a different color. One have to admit that looking at the picture, no advantage of a 3rd dimension can be experienced. The transparent colors, marking the space around a compound state, do rather disturb the recognition of the structure. Furthermore, the transition from State 1 to State 2 is hard to recognize too. In figure 9.8b an image, rendered in Blender, is shown. There are also three compound states, which contain the following states: {State 2}, {State 3, State 3b}, {State 4} (barely visible). Each compound state is enclosed in a colored glass box, while atomic states are light emitting objects. Again, one has to admit, parsing of this image is a cognitive challenge.

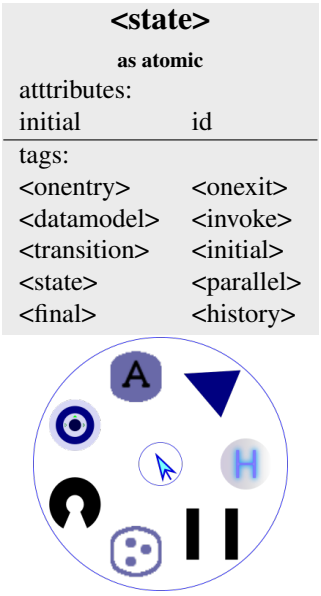
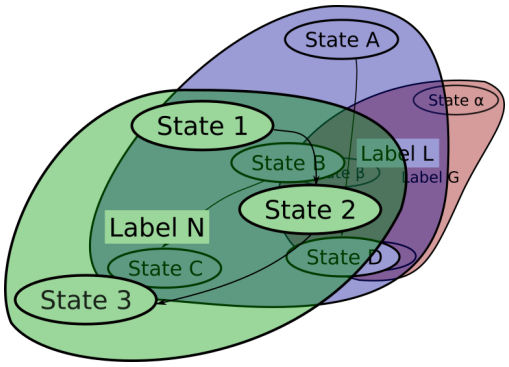
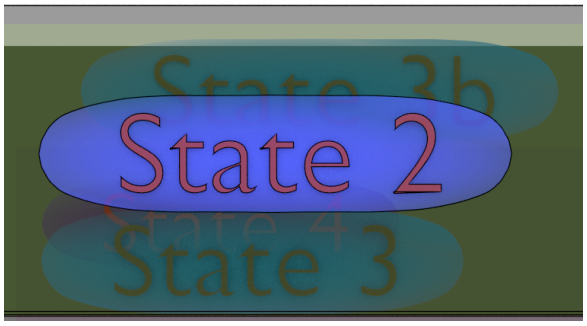


Figure 9.7: Context menu to edit a compound state.

But there is a lot room for improvement. For example, the emitting light can be adjusted depending on the objects distance to the camera, making the appearance of the closer compound state clearer. Sibling compound states should become equally visible if they not overlap, otherwise, the one in the back should be tarnished.



(a) Three overlapping compound states.



(b) Mockup of three compound states in Blender with glass and light emitting material.

Figure 9.8: Mockups for compound states.

To edit a compound state it is equipped with a context menu illustrated in figure 9.7. Its icons represent the following options, starting with the blue triangle running counter-clockwise: new transition, new history child-state, new parallel child-state, new compound child-state, source editor for source tags (which is the same for atomic state, see figure 9.5 on page 82), new final child-state, new child-atomic-state.

In an SCXML-document, a compound state's initial state is either the first child-state in document order, referenced by an initial attribute or being targeted by the transition of an initial tag. But in a graphical environment, a visualization is needed so that the user can operate on it. For that reason we introduce in section 9.0.6 an initial state icon, which is displayed independent on how the SCXML-document specifies the initial state.

The transitions' routing is discussed in section 9.0.8 and the label placement, which names the superstate and is also the anchor for the menu, is presented in section 9.0.7.

In chapter 10.4.2 we present the development of the compound state.

Parallel states

Parallel states are basically, closely cumulated compound states. Imagine four SCXML regions drawn as 3-dimensional boxes, side by side, each containing a nested compound state. By now moving to the side of the cumulus and viewing to its center, it is easy to recognize that distiction of single regions becomes difficult, if not imposible, during exploration. From here follows, that parallel states needs to have a more flexible handling.

<parallel>	
attributes:	id
tags:	
<onentry>	<onexit>
<state>	<parallel>
<datamodel>	<invoke>

One way, for example, is an arbitrary number of windows, which can be defined by the user. A window contains a region of a parallel state and is billboarded screen aligned. The user determines which regions are to be presented in what window. Furthermore, a window is only a 2-dimensional plane, a viewport into the 3D space of the parallel state. That way, a user can configure what to focus in a scalable way.

Unfortunately are our resources to sparse to investigate this topic further.

SCXML (root) state

The SCXML root tag is equivalent to a compound state and streated therfore similar. Only the options in the menu and edit dialog are adjusted.

<scxml>	
tags:	attributes:
<state>	initial
<parallel>	name
<final>	xmlns
<datamodel>	version
<script>	datamodel
	binding

Labels in composite states

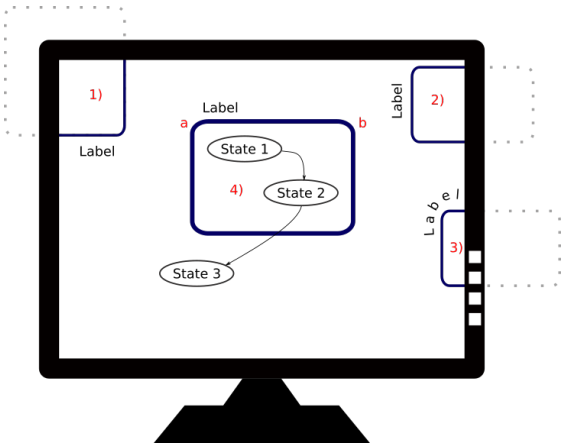


Figure 9.9: Each element's id should be visible at all times on the screen, if the element is visible.

Positioning labels beloning to a composite state is a complex issue. Imagine a nested composite state, which has several children, also being composite states. When one moves further into the first state, then the label should still be visible all the time, without jumping around. Furthermore, if composite state moves closer to the border, the labels should not leave the screen. Figure 9.9 shows some example of intelligent label placement. Considering the state in the middle of the picture, marked 4). If the camera turns to the right, then then label could be moved smoothly, as indicated in 3), to the position 2); ensuring its visibility on the screen. Unfortunately, there is also a catch. There is no smooth movement for the label from position 2) to 1). We can either jump, or flip the label when it is in the middle of the screen.

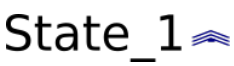


Figure 9.10: Fold a state by pressing the icon.

To save space, composite states can be folded, so that all children disappear and only a place-holder-state stays back to represent the composite state. This is named folding and discussed in section 9.0.7. It is triggered by pressing the icon besides the label, see figure 9.10.

Folding

Figure 9.11: A folded composite state.

Folding is a typical concept for hierarchical organzied items to save screen space by hiding descendent elements of a superelement. When a composite object is folded, then not only all descending states have to be set to *inivisible*, also there transitions have to be re-routed. And not just the outgoing transitions, also the incoming transitions. The opposite operation, unfolding, opens all descending composite states and re-route transitions according to the SCXML-document. Because we expect that a folded state has more transitions, we reserve more space for state, see figure 9.11. Also, the state is equipped with a button for the unfold action, while the button for the fold action is presented in section 9.0.7.

9.0.8 Transitions

<transition>
attributes:
event[n], cond, type

SCXML's transitions can become very complex elements. One transition may have numerous targets, triggered by an arbitrary number of events which may depend on a *cond*itional attribute. That not enough, transitions are also typed as either *internal* or *external* and may carry executable content. The complete logical background is given in 2.4.5.

For a first implementation, we make some limitations. Because several events are separated by a comma in the SCXML standard, we consider here a single line to display events only. That leaves the burdern to differentiate between events by the user. Furthermore, we ignore multi-targeting, which eases the routing problem. The remainings are an informative field for event-id, condition and type as well as a single line connecting the two states, whith its direction indicated on one end with an arrow.

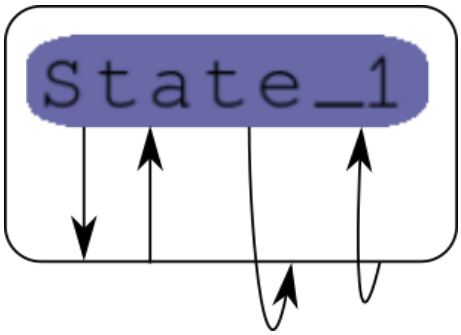


Figure 9.12: Left: internal, parent's exit hander is not called - Right: external, parent's exit hander called.

Showing the executable content constantly would fill the screen in a rather impractical manner. Hiding the content without any indication of its existence on the other hand, would reduce readability of the graph. One would need to open the transitions's code, in whatever manner, only to find out there is none. Therefore an indicator to mark the present of a condition as well as accessor to the editing window is added. We considered several indicators: the transitions color, an additional icon (maybe an asterisk in front of the event label), the transitions stroke style (thickness, dotted, segmented, etc...), a different start/end marker or an indicator on mouse hover. Currently, we ignore the executable content.

Figure 9.12 gives an example how internal and external transitions can be represented visually. This information is currently ignored.

The implementation of transitions is discussed in chapter 10.4.3.

9.0.9 User Navigation

Moving in 3-dimensional space comes with a great level of freedom. We use the standard control capacities. That is, the keyboard is used for translational movement like forward (w), backward (s), upwards (e), left (d), while the mouse, if its left button is pressed while hovering over the background, rotates the view around the x-axes as well as the y-axes. We name the state without a button pressed *pointing* mode and the state where the mouse button is pressed and rotation occurs *explore*-mode.

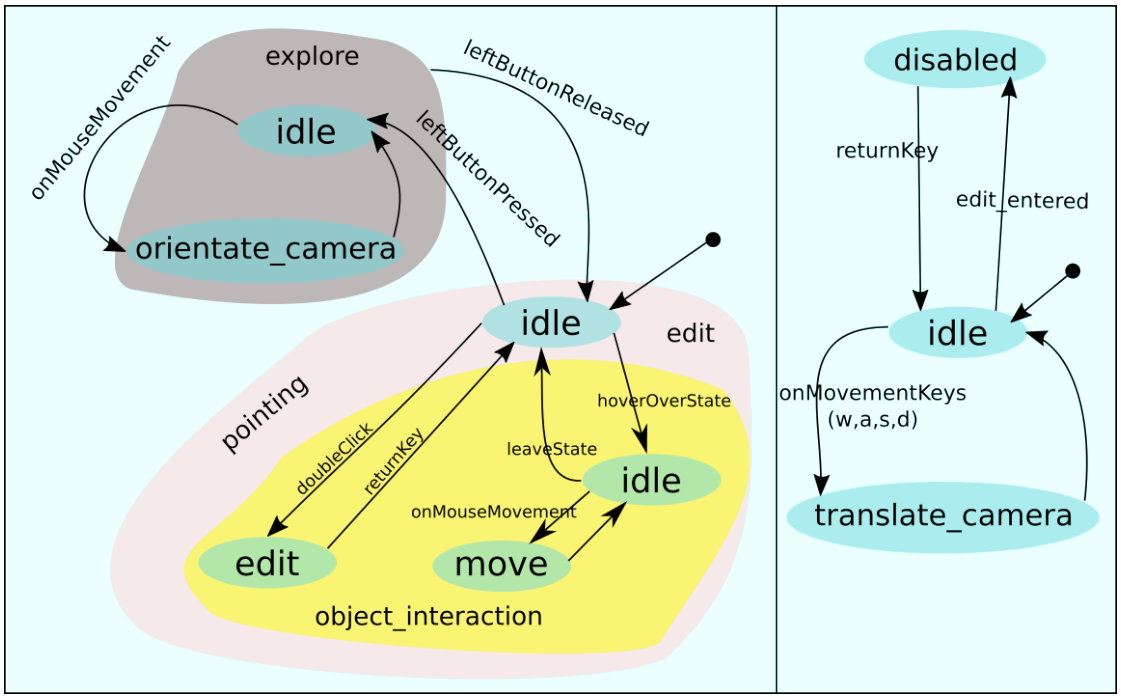


Figure 9.13: Two orthogonal states handle user input. Mouse input is controlled to the right, while keyboard is reflected on the left.

Because the subject changed frequently during development, we state in figure 9.13 our last preference. There are two orthogonal regions, the mouse handling on the left side and the keyboard handling on the right side. Toggling the left mouse button switches between *explore* and *pointing* mode, which are both expressed as compound states. When the user starts editing a state via double-clicking a state-label, camera translation is deactivated (right region). This prevents abrupt camera translation whenever a keys is pressed that belongs to the set of movement-keys and to the new object identifier. Pressing the RETURN key frees the camera translation again, while also the edit state is left.

Arcball

Initially in this subsection we stated that the mouse movement is projected into camera rotations around its local x/y-axis. The Arcball[Sho91] input method calculates from the distance travelled by the mouse pointer a third rotation, which can be used to rotate around the z-axes. The idea is to project the start point and the end point onto a sphere. Afterwards, two vectors, starting at the sphere's origing and ending at the end/start point can be calculated. The angle inbetween represents a rotation around an axes running through the sphere's origin, perpendicular to a plane consisting of the three points: start-point, end-point and the sphere's origin.

Usually, the arcball technique is used to rotate objects, which are observed and presented in 3D space. We experimented with the arcball by setting the sphere in the middle of

the screen, making the camera the origin of the sphere. When the camera was adjusted according to the rotation of the sphere, we were rather confused because there were no visual anchor points. That made it hard to understand the rotation occuring. The experience was surly influenced by the states, which billboarded to the camera.

3D Mouse

A 3D mouse is an input device created for faster navigation and handling of objects presented in 3-dimensional space. It allows to: pan (left/right, up/dow, forward/backward), tilt, spin and roll.

We connect such a device in chapter 10.3.2.

Camera rotation

We experienced the camera rotating around the line of view when using the 3D mouse, the arcball technique and throught the page up/down keys. This kind of rotation brings more confusion than benefits at this point of development. For that reason the camera rotation around the z-axes has been disabled.

Chapter 10

Implementation in C++

In the previous chapter we gave some ideas on how widgets can be displayed in a 3D enviroment and how the user can explore this environment. In this chapter, we argue first for the choice using Qt and show subsequently our attemts to fullfil our priviously defined objects.

10.1 Introduction

Nowadays are many programing languages with powerful development tools available. A complete analysis of what development environment is the right choice for a specific project is probably impossible for non-trivial challenges. Nevertheless, it is still a good idea to argue for the choice one makes. Our choice for using Qt is justified through the following arguments:

- Qt offers native support to read and parse XML files. Although, we evolve some objections of Qt's new way of handling XML, as argumented in chapter 10.5.1.
- A module in Qt[Com16c] provides C++ classes to construct state machines explicitly based on SCXML. Unfortunately, support was halted while SCXML was still envolving, leading to a gap between SCXML and Qt's state machine. We discovered this discrepancy during development and switched to the library **uscxml**. To prevent the editor from being to strong attached to **uscxml**, we created a wrapper library *scxmlInterpreterManager*, which can handle several instances of different state machines. Section 10.5.4 outlines this library. Nevertheless, since SCXML became a *recommendation*[1] Qt resumed developing their SCXML implementation and have an up to date version in Qt 5.7.
- Qt's **qt-creator** is an extensible IDE with standard functionality like file handling, document handling, window management, mine-type recognition, etc.

[1] SCXML is developed by a group inside the W3C consortium, which gives the property *recommendation* to standards worth deploying.

- Qt has built-in support for OpenGL. We learned that the support is incomplete. A wrapper library, **helper_gl**, encapsulates therefore some functionality needed. Section 10.5.3 describes the details.

Besides these facts, Qt is a well known framework, which is used in industry and in a huge number of open source projects since a quarter of a century.

10.2 Navigation and Interaction

10.2.1 Frustum determination

The frustum is the basis for the projection matrix PM and is described in chapter 3.1. Its defining parameters are field of view, near/far plane as well as aspect ratio.

The general approach is to use the window's dimension to define the right and top plane. Based on the fact that the width and hight in NDC is 2, one may find that something between 4 and 8 is a reasonable distance from near plane to far plane, depending on how much space one wants to capture in view direction. The distance, camera to near plane, depends on how huge an object shall be allowed to appear. Textures, displayed with a distance of 1 and no zooming or scaling, appear in their original size. Choosing a near plane smaller 1 allow textures appear larger, while higher values for the near plane will deminish the maximal size. Before we determine concrete values for the PM, we take a step back and contemplate the objects to display.

Pie menus, as discussed in chapter 9.0.4, are rendered in a constant distance to the camera and are therefore not affected by perspective projection. Context menus, relating to floating states, are not affected by the PM either because they are also drawn fixed distanced.The handling of context menus belonging to stationary states is different. Here, the menu becomes part of the state, see chapter 9.0.6. Nevertheless, the menu drawn depends on the rendered state, which makes it only indirect affected by the PM. Transitions are drawn out of gradient textures and an arrow head as end-marker at the transition's destination. Since the curve for the transitions is calculated inside a GS out of supporting points, a normal is needed to calculate a rectangle to catch the gradient and arrow head, see chapter 9.0.8. Because these calculations are performed in screen space, the aspect ratio must be known to determine the normal. All remaining object types are devided into floating and stationary states. And since floating states depend on stationary states, see chapter 9.0.1, we conclude that we just have to consider those together with the transition when specifying the PM.

Stationary states depend on text. Text has a fixed size, measured in pixels. In chapter 10.5.2 we argue for rendering text by a GS directly in screen space, instead of having vertices in global space, transforming them into screen coordinates. If we consider the general approach, we can encounter ill-favoured side effects.

Assuming a PM with the procedure described above and the viewport is set equivalent

to the window's size. When the space is filled with states, their positions are determined in global space, while their dimensions are always calculated in screen space. If the window is resized, and with it the view frustum, which eventually updates the PM, then the states are moved either horizontally or vertically. That is, because the aspect ration changes, which is not reflected in the actual size of the text object. We could pass the changes to the text objects. But that would resize the objects, an affect we would rather avoid.

Furthermore, changing the aspect ration, must also be reflected whenever a normal is in screen space. That is the case for each transition and especially for their arrows.

A simple solution is to have a fixed quadratic projection plane, big enough to capture the larges possible window size. That way the aspect ratio in always constant to 1. The downside of the method is, that the centroid of the projection plane is not the centroid of the actual window. But glViewport takes a parameter to set an offset, which aligns the projection plane's centroid with the centroid of the window.

10.2.2 Object picking

Object picking, *mouse picking* or just *picking* is the procedure to discover which objects reside under the mouse cursor's position. Since OpenGL is not aware of input devices, it is the application's responsibility to map the mouse position to the underlying objects. Nevertheless, OpenGL can assist by delivering the depth and color value of a specific window coordinate. The mouse pointer's coordinate is retrieved from the windowing system.

After a short presentation of common picking procedures, we argue for the procedure we have chosen in respect of a list of requirements.

The following three methods are well known and postulated on websites and literature:

- The **depth buffer**[Mov13, p. 72] method evaluates the distance between the camera and the object to detect. The first step is therefore to retrieve the depth value of the pixel under the mouse cursor's position. A depth of 0 indicates an object on the near plane's position while 1 signifies that there is no object under the pointer. A value inbetween forms in conjunction with the window coordinates a triple that can be unprojected to world coordinates. The object in question is the one with the smallest distance to the calculated position.
- Specifying objects through a unique **color**[Mov13, p. 74] is another possibility to reveal an object located at the mouse pointer's position. The main idea is to assign each object a unique color value and render the scene into an off-screen buffer where each object is drawn with its unique color. Reading out the color under the pointer's position leads then to the desired object. The background color represents the absence of an underlying object.

- **ray casting**[Mov13, p. 76][Ger15] is a solely mathematical way that does not include any interactions with OpenGL. The cursor's window coordinates are transformed twice into world space, resulting in two points P_0 with $z = 0$ and P_1 with $z = 1$. With these points a ray $V = P_1 - P_0$ is formed; connecting the near plane with the far plane. All drawn objects intersecting that ray are then collected in a list as candidates. The candidate closest to the near plane is the object in question.

To choose a proper method for our application we assemble a list of requirements:

1. As accurate as possible. Activating the wrong object on mouse click reduces user satisfactory parlously.
2. Single object detection. It is sufficient if only the closest object to the camera can be detected.[2]
3. Reasonable fast. We want to highlight elements when the mouse hovers over it. Object picking is therefore triggered on mouse movement and not only on mouse clicks.
4. We want to identify specific regions of objects. For example, a compound state can be moved or unfolded depending where on the state the mouse click occurs.

Assessing the presented algorithms is done with these pickable objects in mind: text-based objects (states, event labels, etc.), transitions (start point, target point and the line inbetween) and a Pie menu (pieces of pie must be differentiated). Also, the Pie menu is drawn at fixed distance.

Analysing each algorithm leads to the following observation:

- The **depth buffer** algorithm's accuracy relies highly on the depth buffer's precisition and on the distance between the near and far plane. Nowadays, depth buffers have a 32 bit precisition, which seems to be enough for our purpose. Therefore, requirement items 1 and 2 are not of any concern. But, having the global position of the object, does not allow fast identification of the object's section the click occured upon. To find this section, one must map the coordinate to an object's local space. We rate items 3 and 4 as medium fulfilled. Item 3, because there are calculations to do on every mouse move and item 4, which is just evaluated on an actual mouse click, because the calculations have to be adapted for each object.
- Identifying objects by there **color** values makes it necessary to render each object in a unique color into an off-screen buffer. OpenGL supports off-screen rendering, named MRT, as introduced in 8.5.3. For item 4 we can argue that each objects region has to be identified and drawn differently, no matter if the item is clicked or not. That is some extra work.
- The **ray casting** method has the same evaluation as the *depth buffer* method, because the calculations are identical, only the way the object is identified differ.

[2] Inkscape, for example, offers the selection of hidden items on layers.

We conclude that the **color** picking method is the right choice for the application. Setting up the drawing of the objects' interactive sections can easily be done because all sections have either a mathematical description or are drawn separate anyway. The Pie menu in chapter 9.0.4, for example, is a disk where its sections can be determined by an angle.

10.2.3 Main and debug menu

This section describes the implementation of the Pie menu introduced in section 9.0.4. Figure 10.1 shows the main menu, whith the mouse pointer hovering over the debug icon, which is *bleached out* to emphasize the option to be selected. While the menu appears when the mouse button is clicked, the underlying action is triggered when the button is released. No action is taken when the button is released in the center of the menu or beyond the blue circle.

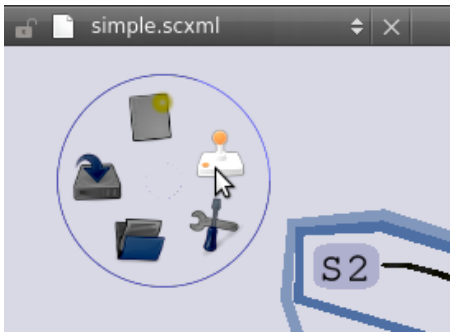


Figure 10.1: Pie menu, with mouse pointer hovering over the debug icon.

The pie menus presented, can be divided into equally sized regions; each region containing an icon, representing an option. We have two pie menus with a different number of regions. The main menu has 5 regions while the debugging menu has 4.

A shader program **ActionDisplayMenu** takes several parameter as input to behave according to the menu is represents. The icon *id* is used to present either the bitmap of the main menu or the debugging menu. This id is retrieved throught the text_3d library, which stores identifiers representing bitmaps, see chapter 10.5.2 and chapter 10.5.6 for more details. To display the menu, the location and the menu's size in pixel is to be defined. For the interaction, like emphasizing the menu hovered by the mouse pointer, we need to identify each region. This can be achieved by 4 further parameters; one specifies an offset where the first region starts, another specifies the amount of the equally sized regions the menu has, while the other two are the radii of the inner and outer circle. Now, only the current position of the mouse pointer is missing. It can be retrieved by the global shader, which is discussed in chapter 10.5.6.

To find the item hovered by the mouse pointer, so that it can be enlightened, a clever interaction between geoemtry shader and fragment shader is necessary. The prinicipal idea is that those fragments, belonging to the icon in question are recognized in the FS and their color treated differently.

To achieve this, the GS issues in addition to the quad, 4 user-defined varyings, specifying a square with the values (1, 1) and (-1, -1), for the left bottom and right top corner, respectively. This defines a coordinate system with its origin at the quad's centroid. The mouse pointer's position lies somewhere inside that coordinate system, even if it is outside the quad. By calculating its angle towards the ordinate, and by having the

number of icons and the offset of the first icon at hand, each FS instance can easily identify if it is part of the section, covered by the icon underneath the mouse pointer. Because radii are also past to the FS, each instance can check, if the mouse pointer is outside the circle, or in the middle circle.

In addition to color the icon, the FS also determines the picking-color. That makes it easy for the system to determine which option (if one) was select when the mouse button is released. Section 10.3.2 discusses how mouse clicks are processed and section 10.2.2 for a discussion about object picking.

An alternative way to identify regions would be a different usage of alpha values per region. Depending on that alpha value, each FS instance could take the apropriate actions (highlight or not). Obviously, the alpha values would not be passed as color value, but very much as picking-color. This method has a performance advantage because it does not calculate as much as our implementation. But performance is of lesser concern here, because there is only one menu and mouse pointer at a time.[3] The advantage our method is that we do not have to deal with alpha values in the bitmap.

10.3 Architecture

The principal architecture of the plugin is illustrated in figure 10.2. There is the Qt-plugin, which is separated into two section. Section 10.3.1 describes how the plugin is integrated into qt-creater, while section 10.3.2 documents the main plugin, handling OpenGL, SCXML, etc.

The four libraries, libtext_3d (visualize text with OpenGL), libhelper_gl (encapsulating OpenGL function for Qt), libscxmlParser (parses SCXML to generate application defined objects) and libscxmlManager (provides access to an SCXML interpreter) are described in the modul section 10.5.

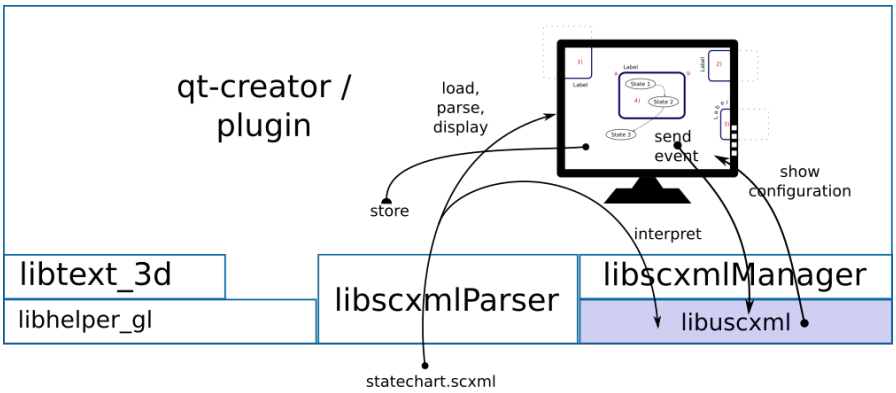


Figure 10.2: Library used/written for in this project.

Figure 10.2 illustrates also a raw workflow. The libscxmlParser reads in a SCXML

[3]Even with two mice, which crossed our minds, is there still no performance issue.

document, which is then passed to the plugin (stateroom editor). The editor displays the statroom and manages the changes applied by the user. Furthermore, the stateroom can pass a serialized version to the libscxmlManager, which then manages the interpretation done by the external component libuscxml. The configuration of liuscxml is displayed graphically from the editor, while the editor can send events to the interpreter. We discuss this in detail in chapter 11. Serializing the stateroom into an SCXML document is elaborated in section 10.5.5.

The remaining part of this chapter describes the aforementioned overview in details.

10.3.1 Qt plugin

Most development for Qt is done in either Eclipse or qt-creator. The plugin system for qt-creator is powerful and allows to take control of a huge part of the IDE. For example, the core elements, *TextEditor* as well as the *Designer*, are implemented as plugin.

For this project we use qt-creator, downloaded from github[Com16b], the version taged with *v3.3.0*. The qt-creator API changes frequently, which makes it necessary to adapt the plugin for higher versions. Although, API changes are not that complicated. But for qt-creator version 4.0, at least Qt version 5.6 is needed, which in turn, does not work with qt-creator version 3.3.

Unfortunately, the plugin system for qt-creator is not well documented. A small tutorial[Com15] helps to get started, but does not show, how to get access to the main screen and implement a new editor.

We took the plugin *BinEditor* as template since it is much simpler then the TextEditor, but offers almost the features we need for our plugin.[4] Figure 10.3 shows the reverse engineered logic and the gate to the stateroom editor. Green boxes represent qt-creator classes (top line) derived for our purposes (bottom line). Dotted lines are event, triggering a function on its receiver, while solit lines signify relations between classes. The three columns express the domains, qt-creator core on the left, derived classes controled by qt-creator in the middle and the stateroom editor on the right. For the sake of visibility, only a single use case (opening a file) is depicted in the figure, which we examine next.

When qt-creator starts, all plugins register them self through the class **ExtensionSystem::IPlugin** which in turn registers a factory **Core::IEditorFactor::IEditorFactory** to instanciate the plugin, which is derived from **Core:IEditor**. During registration, the plugin manifests the filename extension it can open. For the stateroom editor, this is *scxml* as well as *SCXML*. If a file is loaded with a matching ending, then qt-creator sends a "plugin needed" to the factory (**StateRoomEditorFactory**). During this call, the plugin can decide if an already existing instance of the editor shall handle the new

[4]The plain qt-creator plugin, on which our application builts upon has the hash: 015d4c21474945be234285acdf7c628b61b53118 in our git repository.

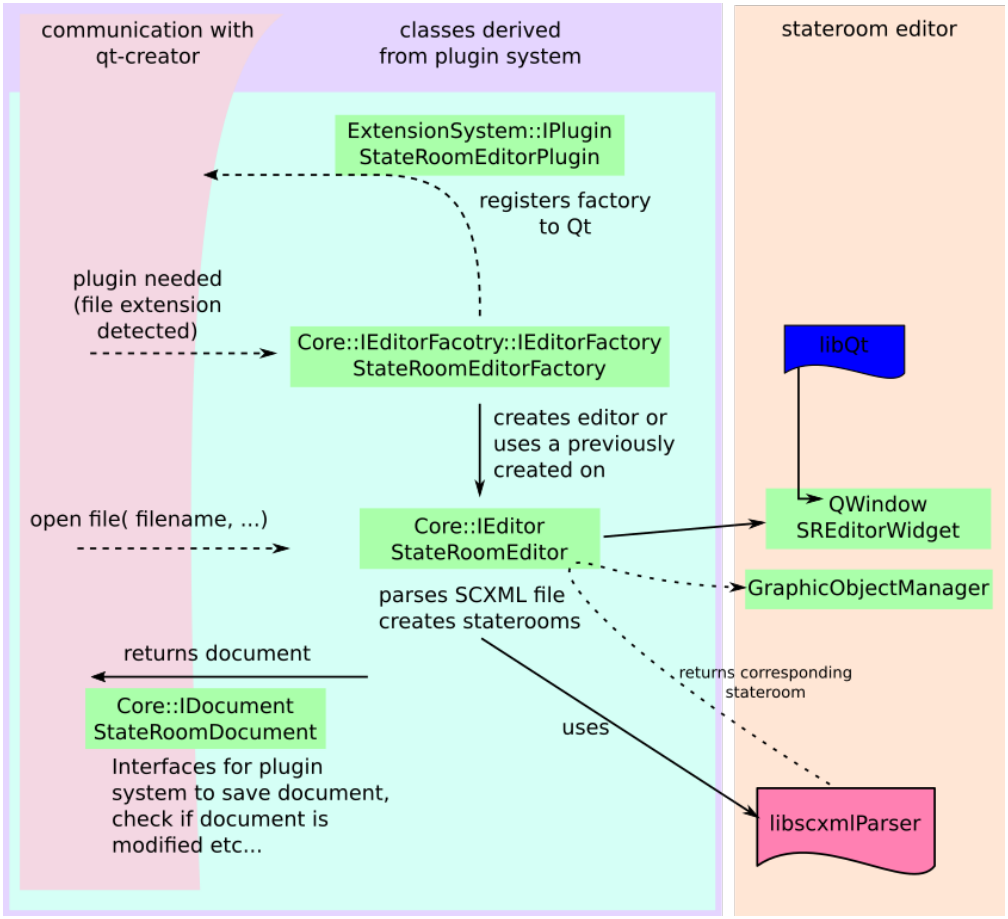


Figure 10.3: This chart shows the reverse engineered architect of a Qt creator plugin.

file, or if a new editor is instanciated. At the current implementation, a new editor is created every time a file is loaded. After creation, the editor's (**StateRoomEditor**) open function is called with the filename as argument. **StateRoomEditor** opens the scxml-file and passes it to **libscxmlParser**, where it is parsed and the corresponding **StateRoom** returned. **StateRoom** contains all objects for the 3D representation, while the SCXML-file is stored in the **StateRoomDocument**.

The library **libscxmlParser** is discuss in chapter 10.5.1 while the **GraphicObjectManager** is part of the editor discussed in chapter 10.3.2.

This stateroom is subsequently passed to **GraphicObjectManager**.

10.3.2 stateroom editor plugin

The core of the plugin is illustrated in figure 10.4. The class **UserInputController** (left upper part) receives all user input; like mouse movement and key press'. It also reads the color below the mouse pointer whenever it moves.

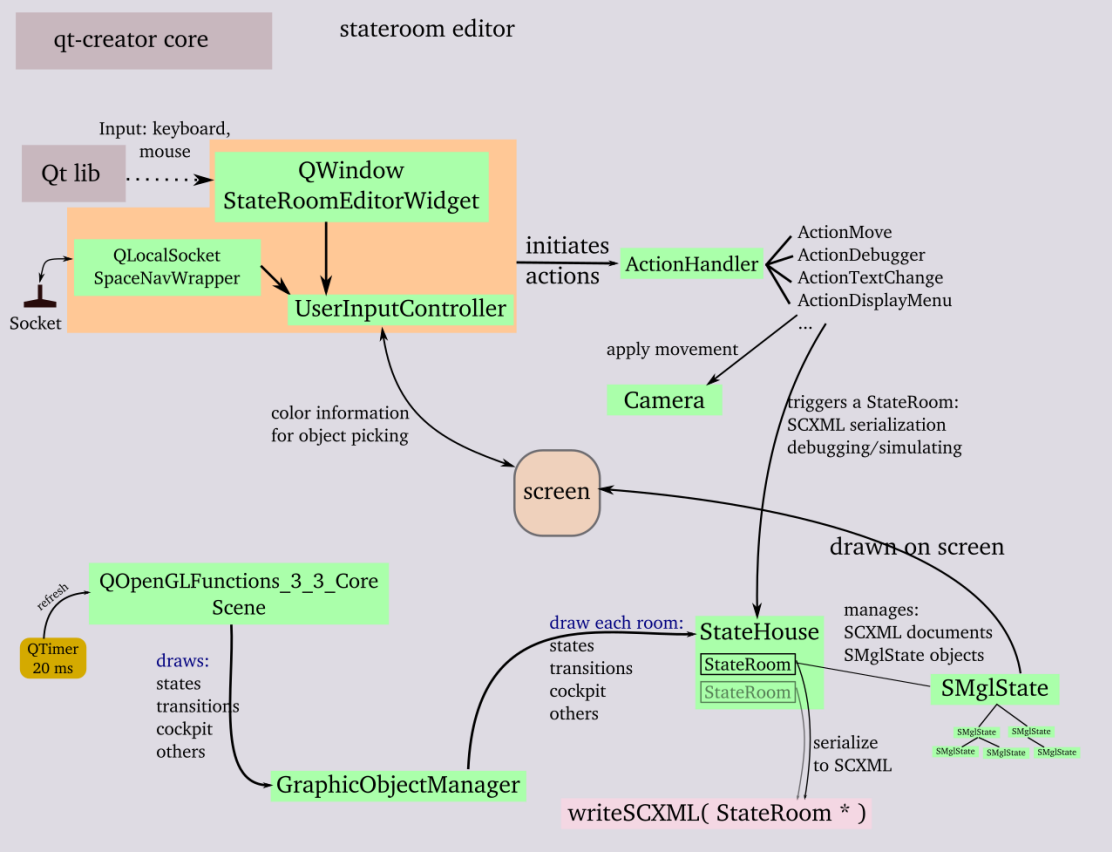


Figure 10.4: The core of the plugin, managing the staterooms.

3D Mouse

Under linux, 3D mice are widely controlled by separate daemons. In our case, the daemon **spacenavd**[Tsi16] manages the communication with the device and offers the information on a local socket. The stateroom-editor handles a connection to that socket in the class **SpaceNavWrapper** (SNW). Whenever a data package arrives from the driver, SNW decodes it and fills an object with the rotation and translation data for each axis. This objects is then passed to the *UserInputController* where it is applied to the cameras position and orientation.

10.4 Modeling SCXML Elements

10.4.1 Stationary States

We proposed in chapter 9.0.6 an ellipse as the visuall representation for an atomic state. But unfortunately, ellipses come with a problem. They cannot not be stretched arbitrary in length to keep the rectangle large enough as background for the text without having a huge overhang on the sides. Stretching the ellipse in Y-direction would help to maintain the rectangle's size, but results in an overhang in Y-direction too. The superellipse [Ago05, p. 463] eases these symptoms by making a compromise between an ellipse and rectangle. It is calculated by the formula 10.1.

$$\left|\frac{x}{a}\right|^m + \left|\frac{y}{n}\right|^m = 1, m \geq 0 \tag{10.1}$$

Figure 10.5 shows the final atomic state. In subfigure b) the menu, appearing when the mouse hovers over quadrant I of the ellipse, is shown. It offers buttons to manipulate atomic states in the scope of SCXML. A window to edit the entry/exit handler, the datamodel and the invoker is poped up when **d**, **i** or one of the green arrows is pressed. The separated indicators inform if a script is present for that function. For example, the figure indicates the absences of an entry handler but the presents of an exit handler. With the yellow arrow head a new transition can be created. It makes the menu vanish while a transition appears, following the mouse until the button is released. This is discussed further in the section about transitions 9.0.8. The **e** in the menu for an extended menu to promote a state into a compound or parallel state.

(a) Atomic state with the menu deactivated.

(b) Menu options: **e**dit, **d**atamodel, **i**nvoker, entry/exit hander and add transition.

Figure 10.5: The atomic state drawn as superellipse. The menu in b) appears on mouse hovering.

Developing the final state took a couple if iterations. A selection is presented below.

Iteration 1

Our first idea is to implementat an infrastructure that can load objects from Blender. Such a facility would free ourself from the burden of managing objects on a vertex level basis.

The C++ **assimp**[Tea15] library facilitates importing scenes generated in Blender. We load a scene we constructed in the Blender experiment. It consisted of the state only

(see figure 6.5a on page 55). Unfortunately, we could not find a way to use the meshes in a more controlled fashion. For example, there is no function reporting the size of an object or how it can be modified.

Using assimp and loading its meshes, would not free us from managing objects on a vertex level basis. Furthermore, using it would add complexity, because the meshes are organized in a hierarchy.

Iteration 2

In the second approach, we calculate the vertices for the ellipse ourself. Figure 10.6 shows the ellipse-model rendered with the draw mode GL_POINTS. To render several instances of the ellipse-model, we use the draw mode *glDrawArraysInstanced*. The *glVertexAttribDivisor* is used to pass each model-instance its corresponding model-view-projection matrix (MVP). That way, each instance is transfromed in the VS stage according to the location and orientation stored in its MVP.

Figure 10.6: The ellipse formed by vertices to draw it as triangle fan.

The implementation has basically two buffers, A and B. Buffer A contains the vertices for the ellipse, while buffer B is filled with MVP matrices. Each time a new ellipse is needed, only an MVP has to be appendend.

In this iteration, no menu is implemented. Ellipses can be picked and repositioned. Identified is an ellipse through the color picking method, described in section 10.2.2.

But we are not satisfied with this solution. Because the ellipse is the background for a state, which has text of different length, it should be stretched to cover the whole area behind the text. But an ellipse cannot arbitrarily stretched in X-direction without becoming sharpend at the ends. Furthermore, the billboarding of a screen aligned object is unnecessary, as we discuss in 10.5.2. This is not a real problem, but it is a flaw.

Iteration 3

To prevent an ellipse from becoming unesthetic sharpend, we divide the ellipse in three parts.[5] The central part, see figure 10.7, can then stretched to cover the background of the text, while the side parts are repositioned accordingly.

The GS drawing the ellipse, renders 3 quads (*left*, *mid*, *right*), appearing as one objects. Two textures are used to show the vertical gradient. The texture to the *right* is mirrored from the *left*.

This method gives us the opportunity to draw different picking colors for the quads. We attached the folding feature, discussed in 9.0.7 for compound states to the *right* quad.

[5]Strictly speaking, the shape lost the privilege of being called an ellipse.

Whenever this quad is clicked, and the state reflects a compound state, all child state's visibiliy is toggled and their transitions re-routed.

Unfortunately is this three-parted ellipse not free from drawbacks. The graphical anchor (GA) does rely on the texture forming the state's body, making their calculation intricated.

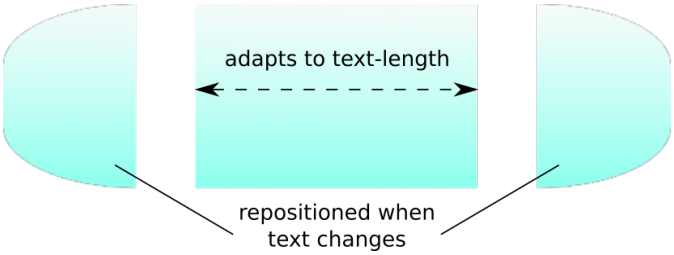


Figure 10.7: Elliptical state drawn out of a three-parted texture.

Iteration 4

Considering the difficulty to calculate GAs for inhomogenous drawn objects, an obvious solution is to draw states with a single function. We experimented with the Catmull Rom[Twi03] (CR) algorithm, using the 4 outer vertices of the text as control points. A state drawn with CR is illutrated in figure 10.8.

Figure 10.8: Ellipsoid draw with the Catmull Rom algorithm; 4 support points are used.

Unfortunately, if the state name becomes longer, the upper and lower deviation becomes much bigger. Practically, the sharpening effect we observed for ellipses, only that is occurs this time in the vertical direction.

10.4.2 Floating States

Compound States

In figure 10.9, a final render of two compound states is illustrated. It is a convex hull from all corner points of each child state. The *rubber ribbon* is calculated by the GS, while the convex hull is calculated with the boost geometry library[Geh+15]. Both calculations are performed in screen space, making the thickness of the ribbon constant.

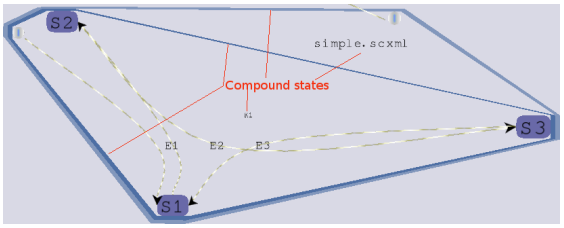


Figure 10.9: Two nested compound states.

The current implementation does render siblings equally. That means, if they are positioned back-to-back, then they will not be recognized as siblings.

Iteration 1

To get a more eye-pleasing appearance for the compound state, we tried to use the Catmull Rom algorithm and display the ribbon as curves instead of having the angular ribbon in figure 10.9.

10.4.3 Transitions

Transitions are drawn with 5 support points and the Catumull Rom algorithm. Two support points are the centroids of the states, independent wether it is a floating state or a stationary state. The 3rd support point the position of the event. The other two support points are the points on the peel of the source, respective target, state.

The calculation is performed in the GS stage. Figure 10.11 shows transitions without a proper event set; making them all pass through point (0,0,0).

Because the transtions are drawn out of several rectangular segments, they have gaps, as in figure 10.10, which are usually hard to observe.

Figure 10.10: Transitions are composed of single rectangles with gaps.

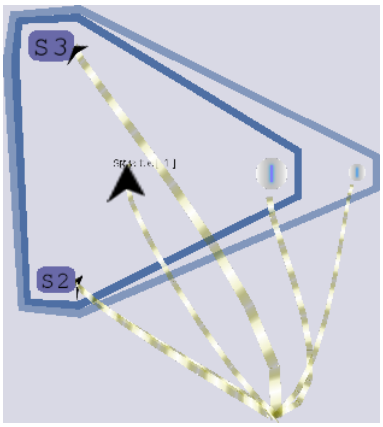


Figure 10.11: No calculations for the transitions' events occured.

Iteration I (GPU)

Figure 10.12: OpenGL's internal lines do not honour perspectivity.

As first implemention we tried GL's built-in line primitive GL_LINE. Figure 10.12 shows this kind of line connecting two states. Each state has a different distance to the camera. It is clearly visible that the line does not adapt it's width when approaching state *S2*, the one closer to the camera. All attempts through GL's API influencing the line's appearing failed. Calling *glLineWidth* with a value greater 1 resulted in a GL_INVALID_VALUE error on both, NVIDIA and Intel hardware. OpenGL's specification states that at least for antialiased lines, widths other then 1 are optional[SA10, ch. 3.5.2]. But no statement for none-antialiased lines were made. So we were *encouraged* to construct lines out of triangles.

10.5 Modules

10.5.1 scxmlParser

This section describes the *scxmlParser*. We omit the actual reading of separate elements but discuss the main loop of the parser. Before we look into the details, we give a small overview of its interface.

The parser's interface consists of three classes and a free function. These three classes, **SMFactory**, **GenericState** and **GenericTransition**, are inherited and implemented by the client application. The derivatives in an application are modelled after there disposition on the client side. In the stateroom-editor, for example, the class **SMglState** (derived from GenericState) is designed to be modified and to be displayed in the editor's OpenGL environment. In a testing application, which we use to construct the Qt state machine, a derivate of **GenericState** inherits also **QState**, making its usage in a Qt state machine possible. Construction of objects like states and transitions occur in a factory derived from **SMFactory**. The customized factory can then establish connections between states and transitions as well as build up the hierarchy of states or simply ignore unwanted functionality. The parsing process is started by calling the free function **buildStateMachine** with an xml document and the customized factory. The factory's functions and these classes build the communication interface between the parser and the application.

Since the Qt-internal state machine is buggy and its development stopped years before the SCXML standard reached its *Recommendation*[6] state, the latter was not pursued until the end of this project.

Qt marked its XML dom parser as *not actively maintained* and recommends the usage of **QXmlStreamReader** to parse XML files and **QXmlStreamWriter** to serialize XML files. We follow Qt's recommendation.

The parser expects a pristine SCXML file. While our extensions from chapter 2.5.5 are recognized and honored, their absence do not harm. Furthermore, the parser expects the aforementioned classes. So, all kind of states, including pseudo states, have to be derived from GenericState, while transitions have to inherite GenericTransition.

The core function of the parser is illustrated in listing 10.1. Before discussing its content, we give a note on the notation. Line 17 calls a function named **create*Element** with identifiers for each state-tag in SCXML. The real source code has for each element a separate function in a dedicated case-block corresponding to the SCXML-tag.

The parser's core component is the function **traverseScxmlStream**, taking a stream-reader **stream**, pointing to the SCXML document, as argument. Because the stream-reader is unidirectional, moves forward whenever an item is read and can not access any item in advance, no information can be accessed behind or in front of the stream.

[6]The W3C consortium marks it's stable specification with Recommendation to emphasize its readiness.

Therefore, two variables, line 4 and 7, are introduced to keep track of the parsing process. There is the **stack** variable, which is a stack of type *GenericState*s. It holds the predecessors while we climb down the XML tree. And there is the *currentState* variable, pointing to the latest generated state object.

The **traverseScxmlStream** function is basically a loop, pulling tags from **stream** until the end of the SCXML-document. Inside the loop is an if-statement, differentiating between opening and closing tags. Both of the if-branches have a switch-statement with blocks for SCXML tags.

```
traverseScxmlStream( XmlStream stream ) {
2 //To keep track of the generation order of states, they are pushed
  //onto a stack upon start-tag and are poped on a closing-tag
4 stack<GenericState> = Stack to store states
  //The current state is the state just generated. It may be
6 //enriched with executable content or become a compound state.
  currentState<GenericState> = NULL
8
    while( tag  = getNextTag() )
10    if tag is start-tag

12      switch tag
          == "scxml" // analog "state" and "parallel"
14            stack.push( currenState )
              currentState = create[State|Scxml|Parallel]Element( sr, currentState )
16
          == "initial" // anaog "final", "history" and "transition"
18            create[Initial|Final|History|Transition]Element( sr, currentState )

20        == "onexit" // analog "onentry"
              absorbExecutableContent( sr, currentState->scxmlState().onExit, "onexit" )
22
          == "invoke"
24            absorbInvoke( sr, currentState->scxmlState().invokeList )

26    else if tag is end-tag
            == "scxml" // analog "state" and "parallel"
28            factory->closing[Scxml|State]( currentState )
              currentState = stack.pop()
30
      end switch
32  end if
}
```

Listing 10.1: Main loop for object construction. After an SCXML snip is read in, control is passed to client application.

SCXML's opening-tags are handled in the first if-branch (lines 12-24) and consist of 4 case-blocks. Whenever a new GenericState is needed, because an SCXML-state tag is read, the first block (13-15) is entered. The *currentState* is pushed on the stack and a corresponding *create*Element*-function is called. All *create*Element*-functions take the *currentState* and the stream-reader *stream* as arguments. Possible attributes of a state are then read from the stream-reader according to the called function. Each create-function reads out all possible attributes belonging to the tag. These are then passes together with the *currentState* to a corresponding function of the factory.

The factory, part of the application, can then create a GenericState-based-objects and fill in the received attributes. It can also build-up the state-tree by using the *currentState* as parent. Afterwards is the new created GenericState returned to the parser's create-

function, which in turn returns it further to the main loop. Here the GenericState substitutes the *currentState*.

The remaining 3 case-blocks handle possible children of state-tags. Functions called in these blocks have no influence of the state-tree and read always until their corresponding closing-tag. That includes also the child-tags, like mandatory transitions for a history-tag.

The 2nd branch of the if-statement (26-29) handles SCXML's closing tags. That is necessary to give the application the opportunity to update a state's configuration. For example, if a state is a compound state or not can only be identified when one knows if it has a <state> as child or not. Since the stream-reader cannot read in advance, a compound state can only be detected after its creation. So, by calling a closing-function, the application can adjust if necessary. A similar problem is the handling of the absence of *initial*-information for compound states. In such a case, the first state in document order is to be treated as initial state. The application could handle that at runtime, by simply checking if that information exists, and if not, refer to the first child. But we need to display an initial state, a real object which can change the target, instead of having a simple tag referencing to a target. Through the closing-function, the application can create an object representing the initial state.

Remark

Parsing an XML file as stream enforces the developer to store all attributes and children one may possibly need during the program execution, unless one is prepared to parse the file again. That means, if a modified version of the file is to be written, then the parser must traverse all elements in the original file, just to dump them in the output file. This is especially cumbersome, if the original file contains XML tags not understood by the parser. Especially because **QXmlStreamReader** does not allow a plain text read from one tag to another, which could then just be dumped to the writer. Furthermore, by parsing and writing a file, written by another program, it is likely that the order in which attributes occure changes. This is annoying if the file is part of a source repository, where histories matter.

10.5.2 text_3d

In various places in our application the visualization of text positioned in space is imperative. For example the naming of states as labels, see chapter 9.0.3, or the naming of events, discussed in chapter 9.0.8. Unfortunately, the open source solutions we encountered do not meet our requirements. As an example we want to name the very advanced library freetyp-gl.

Freetype-gl delivers amazing high quality text. This is not supprising because it uses the ubiquitous libraries freetype and harfbuzz. Both are used in nearly every open source program with the need of displaying text. Freetype delivers withal the actuall glyphs; which are then layed out and antialiased by harfbuzz.

But freetype-gl comes with some caveats. Firstly, no documentation is provided. The demo files, on the other hand, shows basic, as well as advanced features. Secondly, the API has is rather complex. We missed a function like *placeText(position, text)* to simply draw text on a specific position. Instead the application has to handle kerning and other glyph metrices by itself. While freetype-gl has very positive responses over the internet, there is no bigger project known using it, even on a halfway professional basis. Furthermore, freetype-gl is incompatible with OpenGL 3.3, has no cursor support nor can it place text in global space (only screen space). A small test revealed also that the integration into the Qt plugin is not that easy and would make it necessary to change the freetype-gl sources.

Glyph is the terminology used when it comes to typesetting, because its metrics and drawing are from high importance. From now on, we use the term character instead of glyph, neglecting its visual characteristics.

This led to the development of the lightweighted libarary **text_3d** which is based on the following requirements:

- Simple handling of single text lines.
- Support for user input. The cursor is a feature of this library.
- Rotating text. Necessary when compound states are partialy visible at the screens' edge, as explained at 9.0.7.
- Coloring, to highlight text on mouse hover or selection. To spare memory, a color for a letter can be chosen from a pallet with 256 entries.
- Scaling is used to simulate a different font sizes; for example for event names on transitions.
- Optional: Multi-text editing. Useful to add prefixes/postfixes for state labels. This is a nice feature to show/hide XML's namespaces.

Since the development of a full featured text library is out of scope of this work, we have to neglect nicer features typography has to offer; like text shaping, kerning, typefaces and even different font sizes. We also omit a discussion about fonts in general because

it would not contribute to the editor. The only important information are that we use a proportional (as oppose to monospaced) font, which means each letter has a different dimension. For example *i* is narrower then *H*. Glyphs and their metrics are retrieved from the open source library freetype[Tur16].

Drawing text on the screen can be done in various ways. For a single static text a texture containing the complete text might be sufficient. Are several text snippets to be drawn, a single texture containing the text may be partitioned into a *texture atlas*. In that way a pair of (x,y) coordinates defines a text cut-out, which can be projected on the screen. This is sufficient for fixed text since no fragmentation occurs.

But the text we want to draw does change. Therefore we go one step further and address solitary letters, as discussed in chapter 7.2.3. That way, each character has to be added to the atlas once only. Unfortunately, are then 4 vertices needed to draw each letter; instead of 4 vertices per text snippet. But the fragmentation problem is shifted from the atlas to the buffer containing the indices for letters. Fragmentation of such a linear buffer is easier to handle then the one of 2-dimensional objects.

So far, we have defined the texture atlas and from chapter 7.2.3 we know that we need a look-up table where each letter's position as well as dimension has an entry. The metrics of a character is inserted into the look-up table whenever a new character is retrieved from the freetype library and added to the texture map.

When a letter is drawn on a quad, then that quad needs to have the same proportion as it has in the texture map (or look-up table). Otherwise the letter will appear stretched in one direction. One way to do this, is to transform the dimension of each letter into global space dimension; making it a quad in its own local space. When the letter is used, its coordinates are transformed into global space coordinates, as discussed in chapter 3.1. While doing this, the quad's orientation has to face the camera, so that it can be billboarded whenever the camera moves, see chapter 3.3. But care is to be taken when chosing the pivot point billboarding occurs around. That is not the center of each letter, but the center of the text as a whole.

While this is an obvious and direct way to display billboarded text, it seems cumbersome to transform screen dimensioned quads into global coordinates just to transform them back later into screen dimensions. That is especially true, considering that the quads are rendered screen aligned and almost all effects of the perspective projections are vanished, which is as desired and argued for in chapter 3.3. Furthermore, if text is moved, then each vertex of each letter has to be translated separately. Knowing the capabilities of the geometry shader points one to the idea that the GS can draw several letters depending on the centroid of the text.

Figure 10.13 shows the interoperation of several buffers, a texture and pseudo shader code to position text in 3D-space. The actual text to display is provided in **TextChunk**s (TC) by vertex buffer (1). Each TC is worked by a GS instance (2). Because a TC has data for up to 8 letters (3), a GS instance can draw 8 letters at maximum. Texts longer than that are assembled together by several TC, making it necessary that a TC

has the amount of letter stored, variable **letter count**, and the position inside the text available, **chunkWidth**. Each letter in the TC is represented by an index to the look-up table (4). The look-up table contains the metrics, to set a letter, and the positions for the caracter-bitmap inside the texture atlas (5).

To calculate the text's screen coordinate (10), its global coordinate as well as the view-projection matrix (VPM) is necessary. The VPM is retrieved through the UBO (7), a UBO shared between shaders as explained in chapter 10.5.6, while the text's global position is taken from the buffer texture (6). The screen position is then calculated inside the GS (9). Because this is the mid-position of the complete text the TC specific offset **chunkWidht** for text longer than 8 letters, must be added to the position (9). As last step, the character metrics from (4) are added to the position *pos* (9).

For each character is a quad submitted to the FS. With each vertex of the quad, is a position of the character inside the texture atlas submitted (8). That way the character is scaled to the size of the quad.

In addition is for each letter in the TC an index for the color pallete (4), which is also passed to the FS (8). This color is used to show a cursor.

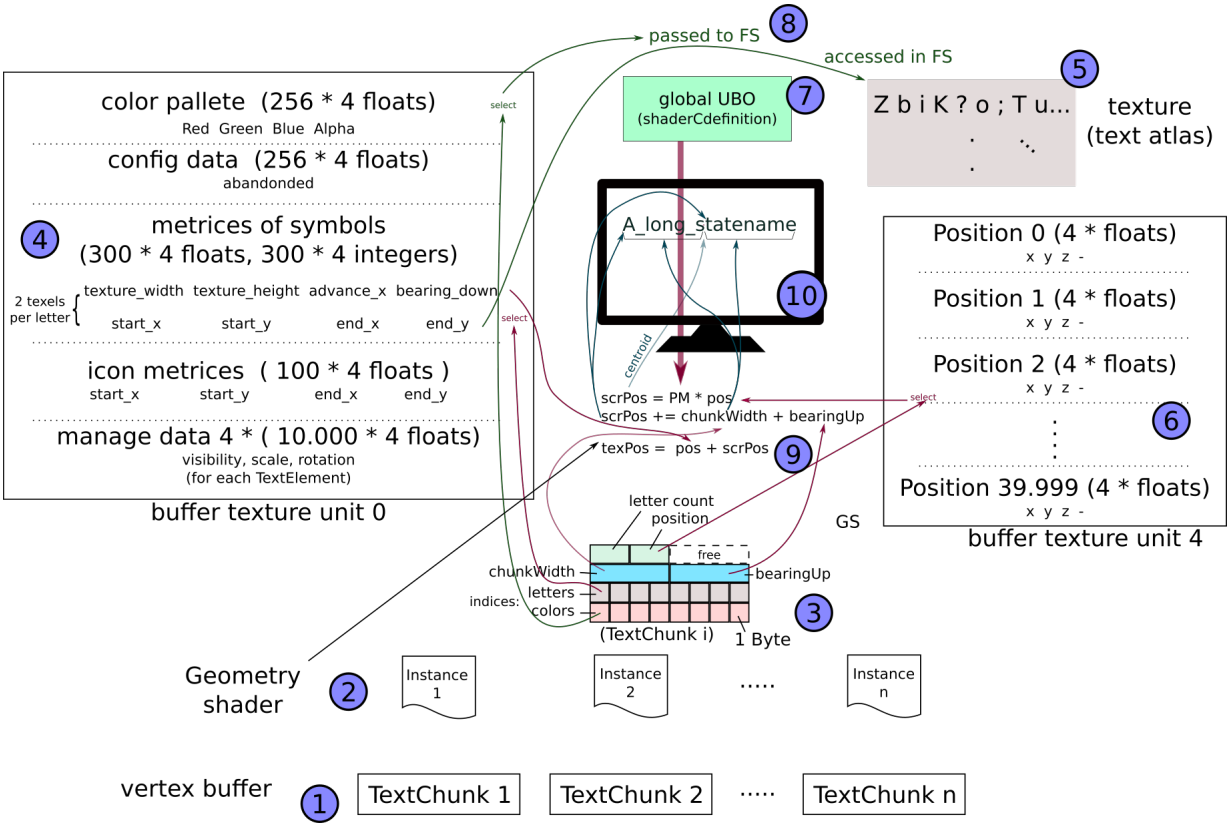


Figure 10.13: The GS to draw text cosumes data from 4 buffers, 1 UBO (global data), 1 VAB (vertex data) and 2 BTO (character data) to gain random access.

The variables *bearingUp* (3) and *bearing_down* (4), and the metrics (4) are not

discussed because they are part of the typesetting procedure and therefore out of scope of this document.

We also omit the discussion on setting up the TextChunks, since it does not contribute to the editor.

To not to issue for each text element a separate OpenGL-draw call, TextElements are assigned to a *group*. Such a group is then drawn by a single OpenGl-draw call. This design is a compromise between granularity of drawing text elements and speed. We use *GROUP_0* for state labels and *GROUP_1* for event labels. By setting OpenGL's *depth buffer comparison* value to *GL_LESS* we ensure that text elements with greater depth-values are overwritten by text with lesser depth-values. Drawing groups in the order *GROUP_0*, *GROUP_1* makes state labels be overdrawn by event labels.

All *TextElement* objects are retrieved by calling the *addText()* function from the singleton *Text3d*. Besides memory management, this singleton builds the gate to some global parameters.

Icon extension

The library has a texture atlas and a buffer object to store the location of texels inside the atlas. For the main application we need a way to store simple icons, like the menu presented in chapter 9.0.4. To make the atlas and buffer object accessible from outside, we extended the interface by a function to add icon. The *addIcon* function takes a filename as argument and returns the entry's position inside the buffer texture. The file is loaded by the *GraphicsMagick*[Gro16] library, which returns a bitmap suitable to be stored inside the buffer.

The ids to use the buffers are shared through a common header file.

Remark

We also tried to integrate a background color. The idea was to draw first a colored quad and render the letters subsequently over the quad. This did not work. No output was generated.

Increasing the number of letters per GS above 8 did not work either and ended in a weired effect. Somehow did, two vertices of one GS instace bleed into a subsequent GS instance, generating a parallelogram between two independent TCs.

and this has an affect on the PM see 10.2.1

10.5.3 helper_gl

Qt offers build-in support for OpenGL. The functions are only accessible through a class derived, in our case, from **QOpenGLFunctions_3_3_Core**. While our *Scene* class derives this class, it feels like a burden to pass it throughout the whole application. Furthermore, are we able to abstract some function calls and free therefore the main application of repetitive source snippets.

We list here only the two most interesting implementations:

- **create_program** takes as argument a vertex, geometry and fragment shader. Furthermore, can 2 string path which are used as preamble for each given shader. That allows easily to prepend the definitions of the global shader data introduced in chapter 10.5.6.
- **BufferWrapper** is a class encapsulating vertex buffer. It handles internally the size of a buffer. If the current buffer is to small when new data is written to it, it automatically allocates a new buffer, copies the content from the old buffer to the new buffer, stores the new data to the new buffer and frees the old buffer. Because vertex array pointer (VAP) dependent on buffer ids, see chapter 8.3.3, all VAPs used to address the buffer, are stored inside the BufferWrapper; allowsing it to re-initialized the VAPs automatically whenever the buffer is switched.

10.5.4 scxmlInterpreterManager

In chapter 10 we state that Qt's internal SCXML based state machine is not up to date. Because for the simulator, presented in section 11, is an interpreter imperative, we evaluated three alternatives. Two interpreters are javascript based, **SCION**[Bea16] and **JSSCxml**[Jun13]. Since both need a javascript environment to work, we compiled Google's V8[coo16] and built a small application to *inject* a javascript program into V8. That way we encountered that both, SCION as well as JSSCXML, need a browser environment to work. Not willing to investigate further, we decided to focused on *uscxml*, a C++ library to interpret SCXML-documents. This leads to the presented wrapper library, *scxmlInterpreterManager*.

Every StateRoom class, see chapter 10.3, has an instance of class **UStateMachine**. This class is used to run an instance of the SCXML-document represented by the the StateRoom. The StateRoom can fire events and retrieve runtime information like: current configuration, reachable targets and possible events to trigger.

The state machine is extended with two I/O-processors, **SocketProcessor**, a class facilitating SCXML's <send>, and **SocketInvoker**, a class implementing SCXML's <invoke>. With <invoke> a socket connection can be established, while <send> uses that socket to transmit messages to a server. The protocol is described in chapter 11.2, where we also discuss the simulator.

A small program (uscxmlManagerDemo) was written to test the library independent from the editor.

10.5.5 Serializing to SCXML

In chapter 10.5.1 we describe how we read and parse SCXML-files. The counterpart is implemented in the free function **writeSCXML** located in the files **scxmltoscxml.{h,cpp}**. It is called by the menu-option Save, introduced in 9.0.4. The function takes a *Stateroom* as argument and traverses the state-tree with the DFS algorithm, which is the only sensible way, since a <state> is to be followed by its children in SCXML notation.

10.5.6 Shared uniform buffer

In section 8.3.1 on page 70 we introduced the *uniform* qualifier how data can be made available to each shader instance through several shader programs. Sharing UBOs among shaders does not just facilitates the upload of uniforms, it also eases adding new variables in shaders for testing purposes. A newly added value is immediately available in all shaders using the UBO. Furthermore, freshly created shaders benefit from the already established infrastructure.

Figure 10.14 illustrates the architecture implemented to facilitate the handling and editing of the UBO we use. The header *shaderCdefinitions* can be included directly in a C++ enviroment, while for GLSL, the library *helper_gl*, presented in section 10.5.3, offers the function *create_program* to mix shaders with other files. Each environment accesses *shared definitions*, a block of magic numbers, which are used in both environment as protocol. Furthermore, each environment has a section in the if-statement. Global uniforms are defined in the section for GLSL and a corresponding setter function is defined in the C++ domain. Updates of variables are done by *globalobjectmanager*, which is discussed in section 10.3.2.

There is also a drawback. Global variables introduced like this are hard to comprehend by developers unfamiliar with the code. That is because they show up without any include directive. When not commented or documented, one may forced to search the source, not just after the origin, also after they way the variable is integrated.

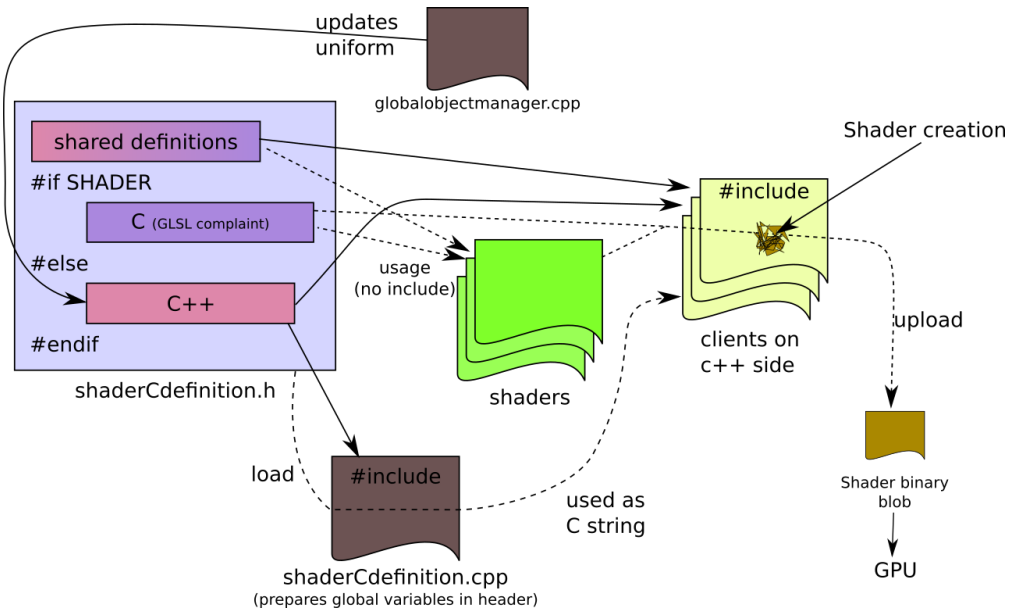


Figure 10.14: Flow of variables to a global uniform buffer, available to every shader program.

Chapter 11

Simulator

In this chapter, we present how a game can be debugged with the stateroom editor.

Harel visualized, in the form of a state chart, the logic of a watch, see figure 11.2. This logic is the foundation of a game developed in BGE. It simulates the hardware of the wristwatch, the kind popular in the 1990s. The logic itself is written in an SCXML file. This file can be interpreted through the stateroom editor and can control the watch through a socket connection.

Because the Blender Game Engine as well as the editor itself were already covered in chapters previous chapters, we refrain from describing implementation details. We state the feature and shortly describe the interaction between the simulator and debugger.

11.1 Harel's watch as blender game

The wristwatch, illustrated in figure 11.1, has a typical 7-segment display for hours, minutes and seconds. Above these segments are indicators for: alarm 1 and 2, beeping (also audible), stopwatch running and an indicator to differentiate between *before* and *after* noon (am/pm). A *wearer*, or rather player, controls the watch by the 4 buttons on it's sides; marked on the figure with **a)**, **b)**, **c)** and **d)**, corresponding with the events on the chart. These are the features defined in the state chart.

Figure 11.1: Digital clock created after Harel's state chart. When a button is pressed a message is send to a connected stateroom debugger.

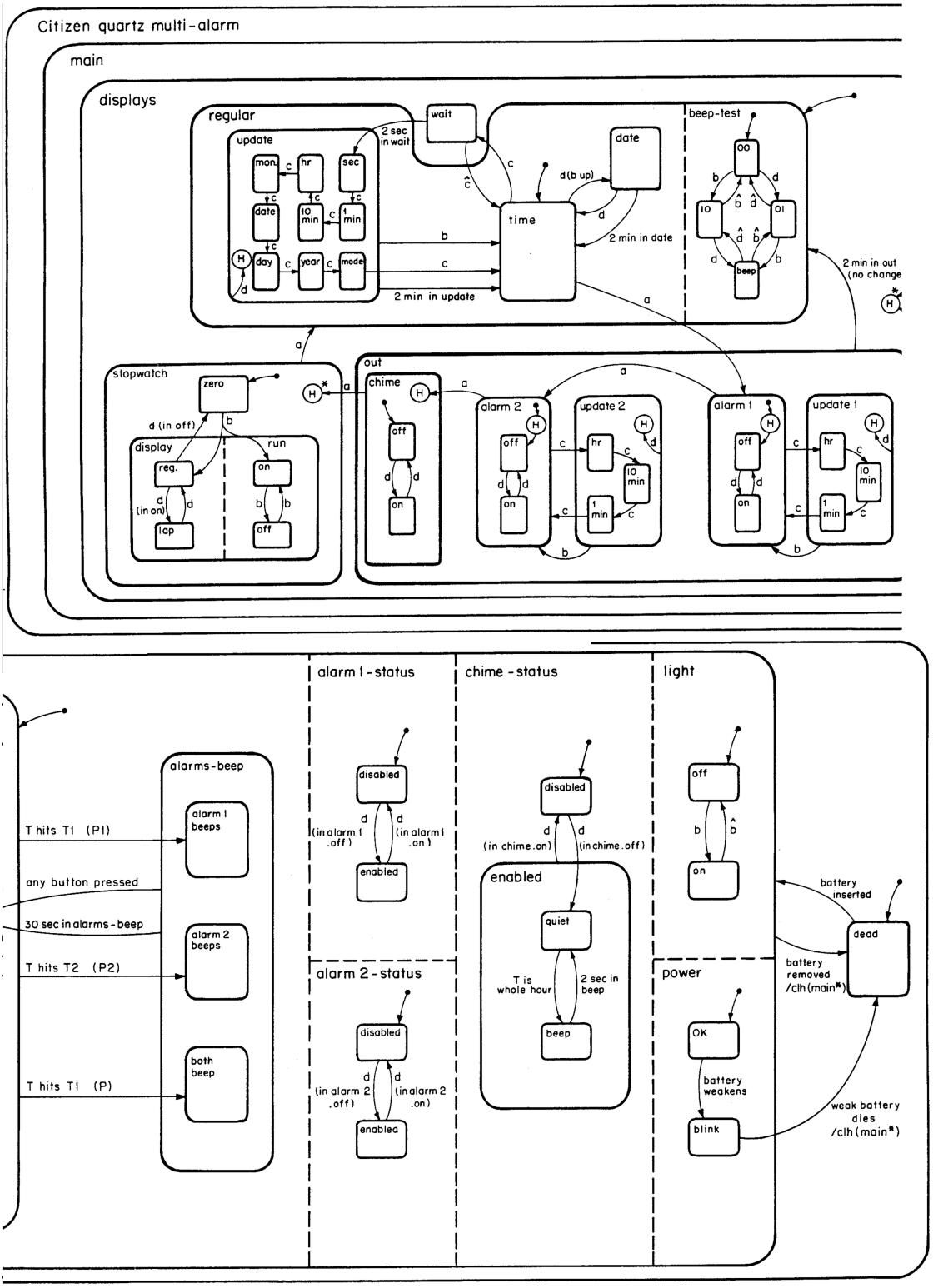


Figure 11.2: Harel's state chart from the mid 1980s describing the logic of a wristwatch. [Har87, p.21 f]

11.1.1 Communication server

When the game commences, a Python script starts a socket server and awaits connections. A client can send commands in form of ASCII text. These commands control the features, like alarm, stopwatch, etc. and are reflected by their indicators discussed earlier in section 11.1.

Figure 11.3: As the original watch, when the light is turned on it becomes harder to differentiated between active and inactive segments.

The messages emitted from the watch are of the form *button_x_pressed*, *button_x_released* and *P*, *P1*, *P2*. Messages are created with the first two sets whenever a state of button changes; they can be mapped to the events in the state chart with: x and $\hat{x}$, where $x \in a, b, c, d$. The latter three occur as asynchronous event when one or both alarms matche the current time. These can be mapped directly to the events in the chart.

11.2 Communication client

The connection betwenn Blender and the stateroom editor is realized via sockets and SCXML's I/O-processor. We use the SCXML tags <invoke> and <send>, described in chapter 2.5.4, while the I/O-processor are implemented in chapter 10.5.4.

Listing 11.1 shows an SCXML snip with an invoke tag (lines 2-5). It connects to the Blender game, see figure 11.5, as soon as the state machine enters the PL.main state, which is the *main* state in figure 11.2 (top). The <send> in line 10-12 will send the command to enable *alarm 1*, if (see line 9) the alarm is current disabled, and the transition in line 9 is taken. Note the id **blenderClock** in both tags. It identifies the connection to the server in the SCXML environment.

```
1  <parallel id="PL.main">
       <invoke id="blenderClock" type="socketConnection">
3          <param name="port" expr="64000"/>
           <param name="ip" location="'127.0.0.1'"/>
5      </invoke>
       ...
7      <state id="alarm_1-status" initial="a1s.disabled">
           <state id="a1s.disabled">
9              <transition event="d" cond="In('alarm_1.off')" target="a1s.enabled">
                   <send type="socket" id="blenderClock">
11                     <content>cmd.setAlarm1</content>
                   </send>
13             </transition>
           </state>
15     </state>
       ...
17 </parallel>
```

Listing 11.1: The <invoke> opens a connection to a server, so that <send> can issue commands to it.

11.3 Debugger

The debugger is part of the stateroom editor. It can be activated through the menu presented in section 9.0.4. Its control window lists events from those transitions that are outgoing from the current configuration. Sending an event is handled by the uscxmlManager as described in section 10.5.4. This module contains the actual interpreter and communicates with the connected simulator through the IO/Processor. After an event is emitted, the system waits until the state machine has a stable configuration. The debugger then reads this information, colors the active states accordingly and updates the event list.

Figure 11.4 depicts a running debugger. The current configuration is **S3** and it's parent. But only S3 is marked since coloring is only implemented for atomic states. A transition with even **E3** goes only out from S3. Hence, the only entry in the list is E3.

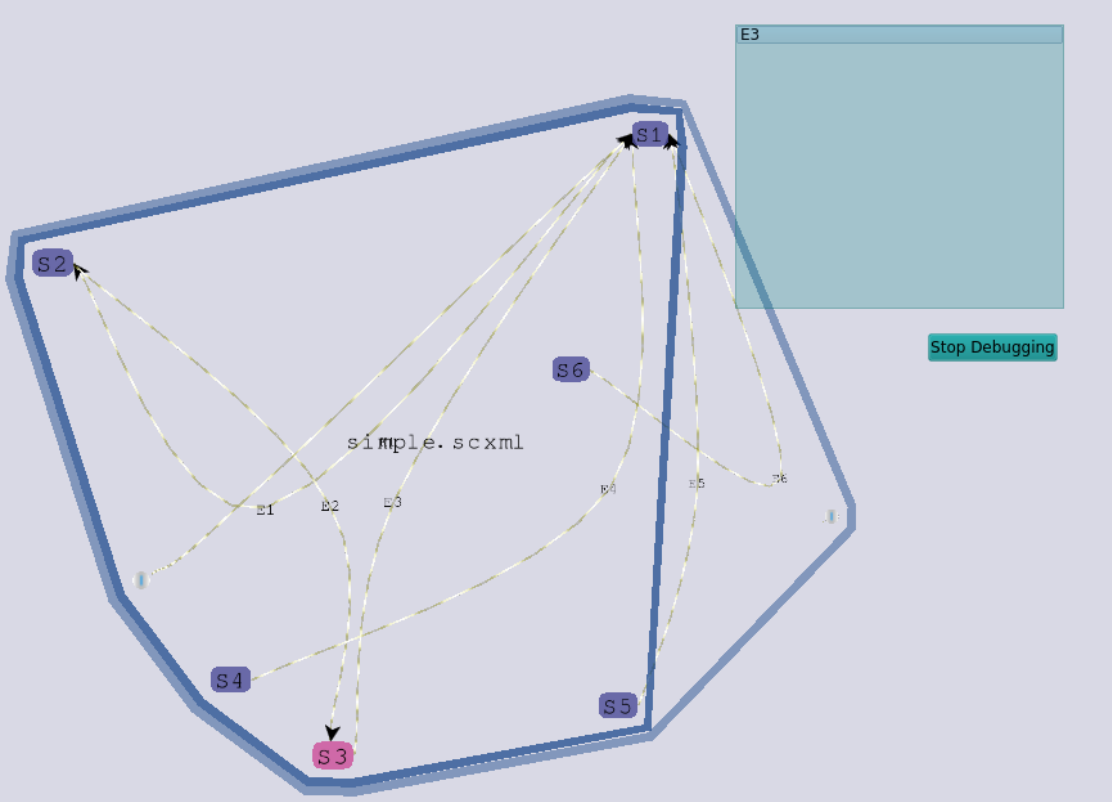


Figure 11.4: Debugging the file simple.scxml, consisting of 6 states. S3 is currently active and can take a transition with event E3.

Harel's watch

The state chart of Harel's clock 11.2 was implemented in SCXML as canonical test during this project. It can be loaded and executed by the demo program uscxmlManagerDemo, mentioned in 10.5.4 to demonstrate the clock. Nevertheless, the stateroom-editor is currently not able to display the stateroom properly as can be seen in the picture 11.5.

One of the most noticeable errors is the weird routing on the left. That is because the algorithm for the intersection of transitions and floating objects is not implemented. It currently takes the position (0,0,0) as anchor point. The twisted arrows and transitions are a second issue, which is caused by Catumull Rom algorithm due to support points, forcing the arc a drastic change in direction.

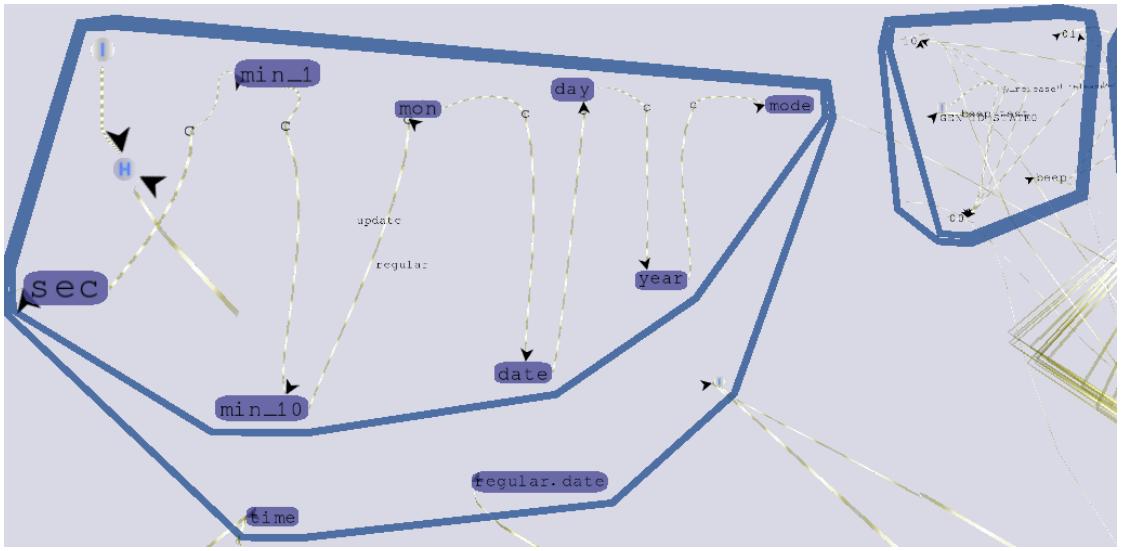


Figure 11.5: Harel's state chart loaded as SCXML into a stateroom. Among the errors: arrows are twisted, transitions are routed weirdly.

Chapter 12

Evaluation and Discussion

"Evaluation": Describe your experiments and analyze your results. Describe what the results mean in detail, and explain why the results are like they are.

Chapter 13

Conclusion

13.1 Summary

We analyzed several software packages to support our idea of having statecharts rendered in 3D-space. Unfortunately, the support for such a project is rather sparse. There is not even a library present to render text in 3-dimentional space.

Anyway, we were able to succeed in displaying SCXML-documents partially. While the results look promising, a lot more reasearch is necessary, wether staterooms is worth pursuing or not. Especially parallale states are a tricky construct in 3D.

Bibliography

[AG15] itemis AG. Orthogonal states. http://www.g-truc.net/doc/OpenGL%204.4%20Pipeline%20Map.svg. [Online; accessed 20-August-2015]. 2015.

[Ago05] Max K Agoston. Computer Graphics and Geometric Modeling: Implementation and Algorithms. 2005, pp. 300–306. ISBN: ISBN-10: 1-85233-818-0. DOI: 10.1016/j.cagd.2009.02.005.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. Real-Time Rendering, Third Edition. 2008. ISBN: 978-1568814247. DOI: 10.1201/b10644-23.

[Art15] Blender Artists. Blender example games. http://blenderartists.org/forum/forumdisplay.php?39-Finished-Games. [Online; accessed 24-August-2015]. 2015.

[Ash14] Kazuyuki Ashimura. The Voice Browser Working Group. http://www.w3.org/Voice/. [Online; accessed 27-November-2014]. 2014.

[Auv15] Mathieu Auvray. Cosmos Laundromat. http://www.imdb.com/title/tt4957236/. Movie. 2015.

[AZ01] Tawfig Alrabiah and Taieb Znati. "Delay-Constrained, Low-Cost Multicast Routing in Multimedia Networks". In: Journal of Parallel Distributed Computing 61.9 (2001), pp. 1307–1336.

[AZ98] Tawfig Alrabiah and Taieb F. Znati. "SELDOM: A Simple and Efficient Low-Cost, Delay-Bounded Online Multicasting". In: Proceedings of High Performance Networking. Vienna, Austria, Sept. 1998, pp. 95–113.

[Bac12] Victor Kuller Baconde. Blender Game Engine Beginner's Guide. Packt Publishing, 2012.

[Bea16] Jacob Beard. SCION. https://github.com/jbeard4/SCION. Accessed: 2016-06-10. 2016.

[Chr11] James Chronister. Blender Basics Classroom Tutorial Book. published under Creative Commons. 2011.

[Com15] The Qt Company. Creating Your First Plugin. https://doc-snapshots.qt.io/qtcreator-extending/first-plugin.html. Accessed: 2015-06-10. 2015.

[Com16a] Blender Community. Logic Editor. https://www.blender.org/manual/editors/logic_editor.html. [Online; accessed 04-July-2014]. 2016.

[Com16b] The Qt Company. Qt Creator. https://github.com/qtproject/qt-creator. Accessed: 2016-06-10. 2016.

[Com16c] The Qt Company. Qt SCXML Overview. http://doc.qt.io/qt-5/qtscxml-overview.html. [Online; accessed 25-August-2016]. 2016.

[coo16] Google cooperative. Chrome V8. https://developers.google.com/v8/. [Online; accessed 27-July-2016]. 2016.

[DH96] Ron Davidson and David Harel. "Drawing graphs nicely using simulated annealing". In: ACM Transactions on Graphics 15.4 (1996), pp. 301–331. ISSN: 07300301. DOI: 10.1145/234535.234538.

[DP11] Fletcher Dunn and Ian Parberry. 3D Math Primer for Graphics and Game Development. 2nd. CRC Press, 2011, pp. 1–824. DOI: 10.1007/s13398-014-0173-7.2.

[Ebe02] David Eberly. Rotation Representations and Performance Issues. https://www.geometrictools.com/Documentation/RotationIssues.pdf. 2002, 2008.

[Fla10] Lance Flavell. Beginning Blender : Open Source 3D Modeling, Animation, and Game Design. Apress, 2010.

[FR91] Thomas M J Fruchterman and Edward M Reingold. "Graph drawing by force-directed placement". In: Software: Practice and Experience 21.11 (1991), pp. 1129–1164. ISSN: 1097-024X. DOI: 10.1002/spe.4380211102.

[Gam+95] Erich Gamma et al. Design Patterns: Elements of Reusable Object-oriented Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[Geh+15] Barend Gehrels et al. Boost Geometry. http://www.boost.org/doc/libs/1_62_0/libs/geometry/doc/html/index.html. 2015.

[Ger15] Dr Anton Gerdelan. Mouse Picking with Ray Casting. http://antongerdelan.net/opengl/raycasting.html. [Online; accessed 20-July-2015]. 2015.

[Gol11] Ron Goldman. "Understanding quaternions". In: Graphical Models 73.2 (2011), pp. 21–49. ISSN: 15240703. DOI: 10.1016/j.gmod.2010.10.004.

[Gro14] Voice Browser Working Group. State Chart XML (SCXML): State Machine Notation for Control Abstraction. http://www.w3.org/TR/scxml/. [Online; accessed 25-November-2014]. 2014.

[Gro16] GraphicsMagick Group. GraphicsMagick Image Processing System. http://igraph.org/c/doc/igraph-Layout.html. Accessed: 2016-08-10. 2016.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. http://www.wisdom.weizmann.ac.il/~harel/papers/Statecharts.pdf. [Online; accessed 27-November-2014]. 1987.

[Har88] David Harel. "On visual formalisms". In: Commun. ACM 31.5 (1988), pp. 514–530. ISSN: 0001-0782. DOI: 10.1145/42411.42414.

[Hec] Aurelio A. Heckert. A Bold GNU Head. http://www.gnu.org/graphics/heckert_gnu.html. [Online; accessed 15-January-2016].

[Hil12] Sébastien Hillaire. "Improving Performance by Reducing Calls to Driver". In: OpenGL Insights. Ed. by Patrick Cozzi and Christophe Riccio. http://www.openglinsights.com/. CRC Press, 2012, pp. 353–363. ISBN: 978-1439893760.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computat 2001. DOI: 10.1145/568438.568455.

[HN96] David Harel and Amnon Naamad. "The STATEMATE semantics of statecharts". In: ACM Transactions on Software Engineering and Methodology 5.4 (1996), pp. 293–333. ISSN: 1049331X. DOI: 10.1145/235321.235322.

[Hop91] Don Hopkins. "The Design and Implementation of Pie Menus". In: Dr. Dobb's Journal 16.12 (1991), pp. 16–23.

[Hub12] Ian Hubert. Tears of Steel. http://www.imdb.com/title/tt2285752/. Movie. 2012.

[Joy15] Ken Joy. Quaterions. https://www.youtube.com/watch?v=mHVwd8gYLnI. [Online; accessed 15-August-2015]. 2015.

[Jun13] David Junger. Client-Side State-based Control for Multimodal User Interfaces. http://www.jsscxml.org/. [Online; accessed 27-July-2016]. 2013.

[KBR10] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL® Shading Language. The Khronos Group Inc, 2010.

[Kin13] Jamie King. Model View Projection Matrices. https://www.youtube.com/watch?v=-tonZsbHty8. [Online; accessed 15-February-2014]. 2013.

[Kra15] Martin Kraus. GLSL Programming/Blender. https://en.wikibooks.org/wiki/GLSL_Programming/Blender. [Online; accessed 27-November-2015]. 2015.

[Leg+12] Eric L.G. Legge et al. "Building a memory palace in minutes: Equivalent memory performance using virtual versus conventional environments with the Method of Loci".

In: Acta Psychologica 141.3 (2012), pp. 380 –390. ISSN: 0001-6918. DOI: http://dx.doi.org/10.1016/j.actpsy.2012.09.002.

[Len04] Eric Lengyel. Mathematics for 3D Game and Programming and Computer Graphics. Charles River Media, 2004.

[Lev10] Colin Levy. Sintel. http://www.imdb.com/title/tt1727587/. Movie. 2010.

[Lina] 3rd Workshop on Engineering Interactive Systems with SCXML. http://www.scxmlworkshop.de/eics2016/.

[Linb] Commons SCXML. http://commons.apache.org/proper/commons-scxml/.

[Linc] Linux counter. https://www.linuxcounter.net/statistics/kernel. Accessed: 2016-06-10. 2016.

[Lind] Qfsm. http://qfsm.sourceforge.net/.

[Line] ScxmlGui. https://github.com/fmorbini/scxmlgui.

[May14] Nikolaus Mayer. Moore to Mealy. CC licence. 2014.

[MB05] T O M Mcreynolds and David Blythe. "Advanced Graphics Programming Using OpenGL". In: Computer (2005), p. 644. DOI: 10.1016/B978-1-55860-659-3.50028-7.

[MF12] Muhammad Mobeen Movania and Lin Feng. "Real-Time Physically Based Deformation Using Transform Feedback". In: OpenGL Insights. Ed. by Patrick Cozzi and Christophe Riccio. http://www.openglinsights.com/. CRC Press, 2012, pp. 265–245. ISBN: 978-1439893760.

[Mov13] Muhammad Mobeen Movania. OpenGL Development Cookbook. Packt Publishing, 2013.

[Ope15] OpenTK. GeometricPrimitiveTypes. http://www.opentk.com/files/tmp/persistent/opentk/files/GeometricPrimitiveTypes.gif. [Online; accessed 24-July-2015]. 2015.

[PNK11] Sergey Pupyrev, Lev Nachmanson, and Michael Kaufmann. "Improving Layered Graph Layouts with Edge Bundling". In: Graph Drawing: 18th International Symposium, GD 2010, Konstanz, Germany. Ed. by Ulrik Brandes and Sabine Cornelsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 329–340. ISBN: 978-3-642-18469-7. DOI: 10.1007/978-3-642-18469-7_30.

[pro09] Tango project. Tango icons theme. http://tango.freedesktop.org/. The official website was not accessible. 2009.

[Rad15] Stefan Radomski. "Formal verification of multimodal dialogs in pervasive environments". PhD thesis. Darmstadt: Technische Universität, 2015.

[Roe80] Henry L Roediger. "The effectiveness of four mnemonics in ordering recall." In: Journal of Experimental Psychology: Human Learning and Memory 6.5 (1980), pp. 558–567. ISSN: 0096-1515(Print). DOI: 10.1037/0278-7393.6.5.558.

[Rou14] Nicolas P. Rougier. Projection Matrix. http://www.labri.fr/perso/nrougier/teaching/opengl/. [Online; accessed 15-August-2015]. 2014.

[SA10] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification (Version 3.3 (Core Profile)). The Khronos Group Inc, 2010.

[SA12] Dave Shreiner and Edward Angle. Interactive Computer Graphics. Addison-Wesley, 2012.

[Sam09] Miro Samek. Practical UML Statecharts in C/C++. Newnes, 2009.

[Sam11] Krystian Samp. "The Design and Evaluation of Graphical Radial Menus". PhD thesis. 2011.

[Sel15] Graham Sellers. ARB_compute_shader. https://www.opengl.org/registry/specs/ARB/compute_shader.txt. [Online; accessed 24-July-2015]. 2015.

[SG06] Inc Silicon Graphics. glBlendFunc. https://www.opengl.org/sdk/docs/man2/xhtml/glBlendFunc.xml. 2006.

[Sho91] Ken Shoemake. "ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse". In: Somewhere 16.12 (1991), pp. 16–23.

[Shr+13] Dave Shreiner et al. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3. 8th. Addison-Wesley Professional, 2013.

[SWH13] Graham Sellers, Richard S. Wright, and Nicholas Haemel. OpenGL SuperBible: Comprehensive Tutorial and Refere 6th. Addison-Wesley Professional, 2013. ISBN: 0321902947, 9780321902948.

[Tea15] Assimp Development Team. Assimp - Open Asset Import Library. http://www.assimp.org/. Accessed: 2016-08-10. 2015.

[tea16] The igraph core team. igraph Reference Manual. http://igraph.org/c/doc/igraph-Layout.html. Accessed: 2016-08-10. 2016.

[Tsi16] John Tsiombikas. spacenav. http://spacenav.sourceforge.net/. Accessed: 2016-06-10. 2016.

[Tur16] David Turner. Freetype. https://www.freetype.org/. Accessed: 2016-08-10. 2016.

[Twi03] Christopher Twigg. Catmull Rom splines. https://www.cs.cmu.edu/~462/projects/assn2/assn2/catmullRom.pdf. 2003.

[Vin06] John Vince. Mathematics for Computer Graphics. 2nd. Springer, 2006.

[Vin11] John Vince. Quaternions for Computer Graphics. Springer London, 2011. DOI: 10.1007/978-0-85729-760-0.

[W3C14] W3C. Homepage W3C. http://www.w3.org/. [Online; accessed 27-November-2014]. 2014.

[Khr15] Khronos Group. Khronos Group. https://www.khronos.org/. [Online; accessed 23-July-2015]. 2015.

[Ope15] OpenGL Object. OpenGL Object. https://www.opengl.org/wiki. [Online; accessed 23-July-2015]. 2015.