DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics Games Engineering

Event-driven Game Engine in Realtime Strategy Games

Johannes Walcher

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics Games Engineering

Event-driven Game Engine in Realtime Strategy Games

Event-basierte Spieleengine in Echtzeitstrategiespielen

Author:	Johannes Walcher
Supervisor:	Prof. Dr. Gudrun Klinker
Advisor:	Sandro Weber
Submission Date:	2017-10-13

I confirm that this bachelor's thesis in informatics games engineering is my own work and I have documented all sources and material used.

Munich, 2017-10-13 Johannes Walcher

Acknowledgments

First and formost, I have to thank my research supervisor Ms Prof. Dr. Gudrun Klinker and Mr Sandro Weber, for the freedom to focus on this research topic that means a lot to me and their assistance and support in the process of creating this thesis.

I would like to show my gratitude to the people of Hackerspace in Studentenstadt Freimann, as well as the development team of *openage*, for the long and fruitful discussions that lead to the problems addressed within this thesis. I would like to thank them for their continuous feedback in the development process. Especially named here have to be Jonas Jelten as the maintainer of openage and Wojciech Nawrocki for his continuous work with the interfacing of the curve logic with the graphics and game-state of openage.

I would also like to thank my family, especially my sister Christiana, for her patience in understanding the problem and her persistent requirement to simplify the explanations.

Abstract

With a traditional lock-step approach in games a unit ordered to move from location A to location B follows the defined path in small steps defined by frames, and the game logic engine checks whether the unit has arrived at location B at every frame. This thesis describes, how an event-based game engine can predict events that happen for example at the arrival at position B. It is therefore required to extend the waypoints of the given path by a time dimension in order to achieve the prediction. The event "arrival at location B" that is generated by the click is not executed as a direct consequence of the click, but is merely scheduled at the time of the last waypoint. When a unit processes the generated event, the state of the game world at the time of the event needs to be in a defined state where the unit has arrived at location B, and all other values have to be valid at the new time.

Beim traditionellen Lock-Step-Ansatz für Spiele-Engines folgt ein Spiel-Objekt, die sich von Position A nach Position B bewegt, dem Pfad in kleinen durch Frames definierten Schritten. Dazu prüft die Game-Logic-Engine in jedem Frame, ob die Einheit an Position B angekommen ist. Diese Arbeit beschreibt, wie eine Event-Driven Game Engine Ereignisse zu State-basierten Zeitpunkten planen kann, wie zum Beispiel der Ankunftszeit der Einheit an Position B. Dies erfordert, die Wegpunkte des Pfades um eine Zeitdimension zu erweitern, um genau vorherzusagen, wann die Einheit diesen Punkt passieren wird. Das Ereignis "Ankunft an Ort B" wird vom User durch Klick indirekt ausgelöst. Es ist keine direkte Konsequenz des Klicks, sondern lediglich das Planen des Ereignisses zum Zeitpunkt des letzten Wegpunktes. Wenn eine Einheit diese Ereignis verarbeitet, muss sich die Spielwelt zum Zeitpunkt des Ereignisses in einem definierten Zustand befinden, in dem das Objekt an Position B angekommen ist. Ebenso müssen alle relevanten Werte auch diesen Zeitpunkt widerspiegeln.

Contents

1 Introduction

Event driven systems are widely spread in most modern programming languages, either as language feature or as libraries [Pyt17][Mic17]. An event is commonly defined as a detectable condition that will trigger a notification ([Fai], page 71).

The origin of event driven design lies in low-level computing and circuit design. In circuit design, events happen in the form of interrupts and are called Interrupt Service Routines [Int16]. An Interrupt Request (IRQ) is for example the interrupt by the network hardware about an arrived package. This IRQ triggers the associated interrupt service routine. The routine is part of the operating system, which in turn propagates the interrupt to the running program. The receiving program can either wait for new data by using blocking `read` [read], or poll the interrupt through polling techniques, such as POSIX-`epoll` [epoll]. The architecture needed to wait for an interrupt may impose difficulties onto system designers, as it usually increases the complexity of the program. A common solution is the creation of network threads, which only wait for the network hardware and are synchronized into the remaining part of the system. Solutions such as network threads increase the complexity and can lead to bugs. This complexity has lead to the development of higher abstraction layers, such as event-based networking, for these kinds of problems.

In modern computer games events happen in the form of user input or internal condition fulfillment: User input is provided through either peripherals or network interactions with other instances, while internal conditions within the game logic (for example a unit enters the attack range of another unit) are fulfilled during the game. An important part of the game logic engine is to check those conditions and trigger the associated events. The traditional approach is to integrate this condition evaluation into the lock-step based engine [Nys14] and trigger the reactions within a frame. Lock-step based game engines use the state of an object in the last frame, such as position and velocity, and calculate the next frame using the time difference between those two frames. A common problem with this approach is the so-called *tunneling-effect*, in which an object may move through another object between two frames, and therefore a collision reaction is not triggered. The *tunneling-effect* is possible to prevent as described in [Fau03] . The resulting position of the two objects is interpolated to the time of first contact. The collision reaction is calculated and the movement of the object is continued into the next frame including the changed path caused by the collision.

This thesis explores the abandonment of the lock-step approach for a division of the game logic into frames concerning the game logic and graphic frames. This method uses lock-stepping for the graphic frames, while frames concerning the game logic are triggered at the time when an event is predicted to happen. The approach is beneficial in games in which the user input is interpreted in form of commands.

Events in real-time strategy games are triggered by the user through commands. The command does not usually take effect immediately [Ont+13]. For example a unit is ordered to move from location A to location B through a click on the map, and a path is generated. With a traditional lock-step approach the unit follows the defined path in small steps defined by frames, and the game logic engine checks whether the unit has arrived at location B at every frame. Within this thesis, using an event-based game engine, it is important that the unit has a defined moving speed which predicts the time of arrival at location B. It is therefore required to extend the waypoints of the given path by a time dimension in order to achieve the prediction. Using this approach, the event "arrival at location B" that is generated by the click is not executed as a direct consequence of the click, but is merely scheduled at the time of the last waypoint. When a unit processes the generated event, the state of the game world at the time of the event has to have a defined state in which the unit has arrived at location B, and all other values are valid at the new time.

For this representation of an event-driven game state it is required to not only track the position of an object, but also its additional properties, which can change during a game. For example, the hit points of an object are tracked in order to determine the time when it reaches zero and stop any movement of the object. It is necessary that there is a timeline with the ability to store the past and the future as well as modify or forget portions of it. This is because some events are triggered by the user, some are executed immediately, such as the deletion of a unit by the user.

The technique used within this thesis is strongly related to key-frame based animation, as used by ChronoCam in the real-time strategy game Planetary Annihilation [Smi13].

The primary goal of this thesis is to determine how a event-based game engine logic is defined using events that are scheduled for executed at specific times and a continuous key-frame interpolation for game-state representation.

2 Motivation

The event-driven approach is not usually applied within internal game engine logic for the physical and logical interaction between objects in the game-state.

Event driven programming provides an intuitive and scaleable solution to design computer system workflows. It is currently largely used within service development, in which different smaller services communicate events and notifications to provide a larger fully featured service. As described in chapter 3.2 and 3.3, many programming languages possess built-in event logic, which are either used to perform remote procedure calls or local procedure calls creating decoupled systems. Decoupled systems allow system designers and programmers to eliminate dependencies from architectures, and focus on developing interfaces. This approach is for example common in end-user applications with input through user masks, in which usually a button-click is bound to a callback procedure. When the user clicks on a button, the window management system of the platform sends the click event, together with the cursor position, to the application. The application subsequently receives the event, and is able to translate the screen coordinate of the cursor position into the respective button and triggers the associated callback. In certain domains, such as large scale web services, it is common practice to use event driven distributed systems if the execution of requests is distributed over multiple computers. As described in chapter 3.2, the python AsyncIO system provides a comfortable approach for a developer to use event based logic, for both input based and internal event sources.

One issue occurring in the application of event based systems in game engines is that most events are not executed immediately. Currently existing event engines are not designed to execute an event at a specific, later, point in time, but to react to incoming events as quickly as possible. Some events, such as the completion of research or damaging of a unit, are generated, but are executed at a later point in time. Within real time strategy games, there is therefore usually a delay between the issuing of the command and the actual completion of the action by the unit. Events therefore always describe the future of the game state. Events defined before the requested frame time are executed before this frame in order to receive a valid game state for the given time. In games,the time when a certain condition could take effect is predictable. But it does contain an uncertainty, if the user changes the state, which might lead to a change in the condition. For example, a unit could receive a new command before finishing to complete another task, such as the construction of a building.

Therefore it is inefficient to calculate the event right after it is generated. Based on the game state at the time of issuing the command, the condition for the event changes, and is recalculated every time a value changes.

With existing event based engines it is necessary to know the full state of the world at the time of the event in order to calculate all effects. It is impossible to know the exact state of the game state in the future in real-time strategy games with user input. In order to represent the past and the future similarly while keeping a full history of the value, curves are designed.

ChronoCam provides a history for any value used in the game state - and the history is calculated into the future [Smi13] With a curve it is possible to interpolate to any specific point in time, even the future, as long as there are enough valid key-frames for this interpolation. This thesis describes how this is used to calculate the state of the world at the time of rendering a frame or for any defined event. The interpolation of the game state makes events to execute at the exact state to operate on and, if all previous events have been processed, it is only needed to calculate the event at the defined time of its execution. For example, a collision event is executed at the exact time of the collision. It receives the positions of the colliding objects necessary with the values already present in order to calculate the bounce [Pro16].

However, it is not enough to only keep track of the positions of the units, since any value that can change over time is important for the game state and that it is tracked by the curve system. Therefore tracked values are for example building progress, speed, maximum damage, or current hit points.

This thesis follows the following goals in order to use this approach in a real-time strategy game logic engine:

- **Continuous representation of gamestate values** of the game state to have every intermediate value.
- **Event driven game logic** to define interactions between objects.
- **Game-time triggered events** to register events into the future.
- **Developer-friendly API** for usage by a large community.

3 Related Work

Event-driven programming is a fundamental pattern in many functional programming languages. However, it has been adopted by other programming platforms. As an example, .NET provides built-in event handling, while AsyncIO was developed for Python3. This thesis extends event-driven programming using a continuous representation of values within the game-state of real-time strategy games, while utilizing technologies borrowed from key-frame based animation systems.

The following sections describe techniques that work closely related to the proposed game logic engine architecture. Existing event driven programming frameworks such as python asyncio or .NET delegates are prominent examples for commonly used event architectures. Time continuous representations of objects and worlds are important for the proposed event framework. They have a long history within key-frame animation, which is used in many games and movies today.

3.1 Environment of Realtime Strategy Games

Real-time strategy is a sub-genre of strategy games in which players build an economy and military power. In this context, economy consists of gathering resources and building a base. Military power comes from training units and researching technologies. The goal is to defeat the opponents by destroying their army and base. They are simultaneous games with more than one player can issue actions at the same time. Additionally, these actions are durative, i.e. actions are not instantaneous, but take some time to complete [Ont+13].

Real-time strategy games are command-driven which means that a player does not directly control the immediate actions of an object: The player gives a command to a unit, for example to move to a certain location, but the unit decides on the path that it will take [Eme13]. Furthermore, if a unit is ordered to attack, the immediate action is merely an animation, and the player has no direct control.

The game-state of a real-time strategy game usually consists of permanent, uncontrollable obstructions and modifiable, controllable objects. Permanent obstructions are for example mountains or rivers on the game map, that cannot change in any way during a game. Controllable objects, for example units, are modified during a game and are controlled by the players Buildings are in between those classes in the context of path-finding - they are obstructions to path-finding algorithms, but can change over time by being added or removed [Eme13] [Pot00]. Units moving around the map are mostly independent, which means once the player has given the command to move to a specific location, they follow their calculated path and no further interaction by the player is required.

3.2 Python AsyncIO

In python, asyncio is used to parallel coroutines [Pyt17] using event loops. A coroutine is a function, in which the execution is interrupted at predefined points, with the code is for example waiting for input using the `await` keyword. The coroutine is resumed exactly at the point it was stopped, containing the newly awaited data. An event loop contains the queue, with tasks that should execute on certain conditions, or on which other input such as network listeners, are registered. The arrival of data is a push of data into the system, which then selects the correct task to execute. The advantage of this framework is that it can run concurrently on a single thread. The coroutines are suspended on `await` and resumed again upon the completion of the the awaited condition. Coroutines save the stack and reload it again later, just as generators in python use `yield` [Pyt17]. The developer benefits from a convenient way of defining asynchronous dependencies, such as waiting for input or output. Other techniques include the setting up of special polling structures such as `epoll` [epoll], or having a separate thread that waits for input in a blocking manner.

The AsyncIO approach is a combination of push-based and pull-based events ([Fai], pages 313-315). If set up to wait for the completion of other methods, it is able to solve interconnected, asynchronous event executions while waiting for arbitrary events. The main tasks are usually configured at start time. This includes waiting for input on a socket. The event loop is also configured at run time by using `await` for a result of an event or by creating a new task.

When used for networking I/O a receiver for a socket is defined. That receiver is called every time that data is sent to the socket. Such an event is for example a new network connection on a server or a file descriptor ready to read data. The system itself has no control over what is arriving and when.

While a method is `awaited`, it executes another task in the event loop and polls the result. The waiting method is resumed after the awaited call is completed.

Listing 3.1 shows an exemplary network client. It defines the coroutine `tcp_client` using the `async def` keyword. This coroutine connects to an open connection using the stream interface, and generates a reader and a writer object.

It further uses the `writer.write` method to send data to the remote server and waits for a line in return. The waiting call is marked `async`, so the execution of the main loop will continue with other tasks. `run_until_complete`, with a coroutine as its argument, is used to start the loop which returns when the given coroutine returns. The loop is finally `closed` in order to clean up used resources [Pyt17].

For delayed execution at specific times, as required for the game engine proposed in this thesis, there are multiple methods depending on whether the method is called periodically or once. The logic coming closest to the one proposed in this thesis is the `loop.call_soon(time, callback)` interface. This method calls the `callback` in exactly `time` seconds [Pyt17]. Events started with the `call_soon` method do not provide functionality to reschedule nor does AsyncIO deliver a game state representation as is required for the approach in this thesis. Furthermore, with AsyncIO it is not possible to reschedule or cancel a given event after it was fired. The logic to reschedule and cancel events is needed in the implementation of each event, generating an error-prone interface. For this reason AsyncIO was not found fully suitable to achieve the desired game logic engine.

Listing 3.1: Example TCP Server using AsyncIO

```
# From Python 3.6.3 Sec. 18.5.5.7.1.
# TCP client sending and receiving

async def tcp_client(loop):
  reader, writer =
  await asyncio.open_connection(
    '127.0.0.1', 8888,loop=loop)
  writer.write('hello␣world')
  line = await reader.getline()

loop = asyncio.get_event_loop()
loop.run_until_complete(
  tcp_echo_client(message, loop))
loop.close()
```

3.3 Microsoft .NET

Microsoft's .NET Framework uses two ways of message delivery models. They both implement the observer-producer pattern ([Fai], pages 155ff).

The producer is the data container storing the data and observing any changes made to the contained data. One usage example is the change-observation of singular types or collections. Singular types, such as integers, use the `Observable` interface, while collections, such as lists, use the `ObservableCollection` interface. A change of the data stored inside an observable triggers the observer. The framework then calls the observer's delegate subscribed to the change of this value.

This approach is implemented for example in the Windows Presentation Foundation, a modern and powerful user interface framework.

The first approach of .NET to implement event callback mechanisms is the untyped object call. This approach uses a delegate to reference the event-callback, which is subsequently executed. A delegate is comparable to a function pointer in C-based languages or method-objects in python. Delegates are stored in attributes using the `event` keyword, which enables the modification of the list of subscribed delegates. When the `event` is called, all subscribed delegates are triggered sequentially [Mic17].

The second approach for event-registration is the typed object call, in which a subscriber object has to implement a predefined interface. This interface defines a method for calling whenever a certain condition is met. With this approach the subscribers are managed using traditional ways of list management [Mic17].

Listing 3.2: TCP Receiver using .NET

```
// Shortened from https://docs.microsoft.com/en-us/dotnet/framework/
// network-programming/asynchronous-client-socket-example
private static void ReceiveCallback( IAsyncResult ar ) {
    // Retrieve the state object and the client socket
    StateObject state = (StateObject) ar.AsyncState;
    Socket client = state.workSocket;

    int bytesRead = client.EndReceive(ar);

    if (state.sb.Length > 1) {
        Console.WriteLine(state.sb.ToString());
    }
    receiveDone.Set();
}
// Create the state object.
StateObject state = new StateObject();
state.workSocket = client;

// Begin receiving the data from the remote device.
client.BeginReceive( state.buffer, 0, StateObject.BufferSize, 0,
new AsyncCallback(ReceiveCallback), state);
```

Listing 3.2 shows how a client is implemented using asynchronous interfaces in the .NET framework. The `ReceiveCallback` are called upon arrival of new data. The function reads data from the socket using `EndReceive`, which will return a buffer of the read bytes. The received data is printed to the receiver's console. The argument to `ReceiveCallback` is used to communicate the result back to the caller, signalling that all bytes have been received. A `StateObject` is used to track the progress of multiple callbacks for the same connection. The receive process is started by calling the `BeginReceive` method with the associated buffers and callbacks.

Scheduled execution is also possible by using the provided method from the .NET framework `System.Threading.TimerCallback`. It is invoked using a timer object which triggers the timer regularly, which in turn triggers the execution of the callback. The callback is call>ed periodically until the timer object is disabled. The period of the call is adjusted while waiting for the timer. The callback itself is dispatched to another thread provided by the system.

The .NET frameworks approach of handling events and notifications is well suited for front end interactions and network stream handling. However, it also lacks properties required for a game logic engine such as the modification of the time until the execution of a scheduled event.

3.4 Key-frame Animation

Animated movements in film sequences are usually generated using key-frame animation and motion capturing. A key-frame is a fixed point in the movement of a joint or object, that defines the path of movement. Motions are generated using interpolation methods such as linear or cubic spline interpolation [FC80].

Key-frames are also set for the whole world at fixed intervals, or only for parts of the model, at relevant times in the motion process, as described in [Stu84]. Fixed intervals give snapshots of the world at regular times, independent of the movement of joints or objects. If it is acceptable to sacrifice some accuracy, it is possible to use only first order linear interpolation [FC80].

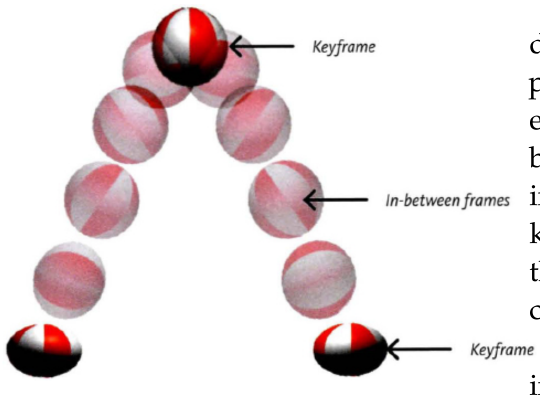


Figure 3.1: Key-frame animation of a jumping ball [Aut09]

Key-frame animation provides a large reduction of complexity for cinematic, completely predefined sequences. This enables efficient compression of movement sequences, but it is not suited for a game logic engine in which interactions are defined as well, as key-frame animation is predefined. However, the general idea is suitable for usage in a time-continuous description of a game state.

Figure 3.1 shows an example of a bouncing ball represented as key-frame animation. There are actually two dimensions of the ball being interpolated, the first one being the x and y coordinate of the ball, which is fixed at the bottom of the bounce, at its highest point and on the bottom of the bounce again. The positions between these key-frames are interpolated using splines, which is apparent due to the irregular distances of the in-between frames. The second dimension that is interpolated is the shape of the ball, which is compressed along the y-axis on impact.

The interpolation method used here is either discrete, as having only this shape at the bottom of the bounce or as asymptotical to the round shape of the ball at the bottom key-frames.

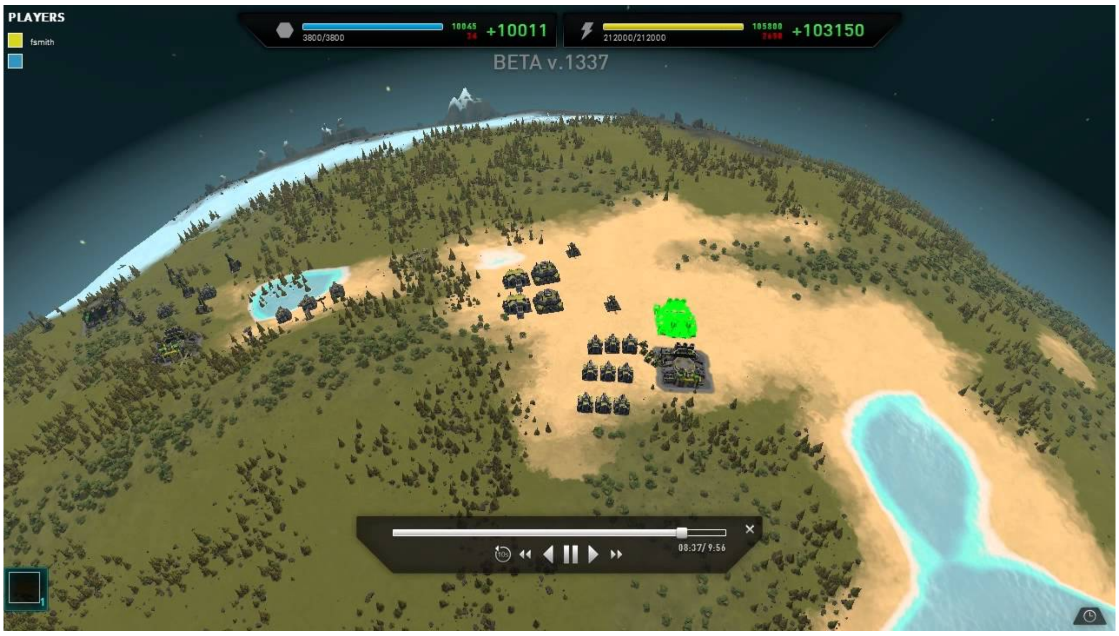
3.5 Chrono Cam: Planetary Annihilation

Planetary Annihilation is a real time strategy game in which players start on different planets, build an economy on their planet and can send troops, or even moons, into other players' planets. It uses a system called ChronoCam that allows the player to view the timeline flexibly, in a way comparable to video player search bars. [Smi13] describes it as:

> "The ChronoCam is similar to a replay system except it's in the live game. While you are mid-game you can jump back to look at the world from any point in time, play in slow/fast motion, scrub the timeline from start to finish, and even play in reverse."

In figure 3.2 shows a screenshot of the running Planetary Annihilation with active ChronoCam. On the bottom of the screen the navigation bar is shown with controls for fast and normal forward and reverse playing. Above the buttons is a slider to show the currently viewed time, which is the filled bar, in relation with the actual game time, which is the full width of the bar.

At its core the ChronoCam tracks each value of each unit via key-frames. The sequence of key-frames in ChronoCam is called curves. This is more efficient than keeping track of every value in every frame, because intermediate values are interpolated based on their surrounding key-frames. The key-frames are usually extrapolated into the future by a short time frame. If the user sends a command, the change is applied at the correct time within ChronoCam, and the following key-frames are corrected. Key-frames also greatly reduce the amount of data for sending through the network for multiplayer synchronisation, since only key-frames and corrected key-frames are transmitted. The physics engine runs at a significantly lower frame rate than the graphics by using curves, but it can generate more than one key frame for every value per physics frame [Smi13]. Furthermore, physics is not required to re-extrapolate from their last prediction, if they still contain valid key-frames.



Uber Entertainment on https://youtu.be/VOas6EFJ9X4

Figure 3.2: Screenshot of ChronoCam in a running game of Planetary Annihilation

ChronoCam is utilized in Planetary Annihilation as a game play feature, in which the player can go back in time and re-view things that have happened. However, ChronoCam is not incorporating a fully event-driven design into the game logic engine. The game physics for Planetary Annihilation runs at a distinct frame rate, not utilizing the predictive capabilities of curves for event execution.

3.6 *OpenAge*: Open Source Real-time Strategy Game

OpenAge [OAge15] is a free and open source remake of the *Age of Empires* game engine (www.ageofempires.com). *OpenAge* was originally designed to make use of the original game assets. Its goal is being more extensible than the original Genie engine, which powers the Age of Empires series. Furthermore, due to its open source character its main platform is GNU/Linux, with a Windows-compatible version currently in the making. Another goal is to test different technologies and use modern language features of python and C++14.

The current game engine is based on a traditional lockstep approach, integrating positions from frame to frame, and executing condition checks during every frame.

The game state currently consists of a container holding every object in the game. Each object is assigned a list of attributes and abilities. These attributes define the available active and passive action and target abilities for a unit. From these attributes actions are activated, for example *AttackAbility* enables a unit to target another unit with *HitpointAttribute*, in order to attack and do damage.

Figure 3.3 shows a screenshot of a running game of *OpenAge* in the version 0.3.0.

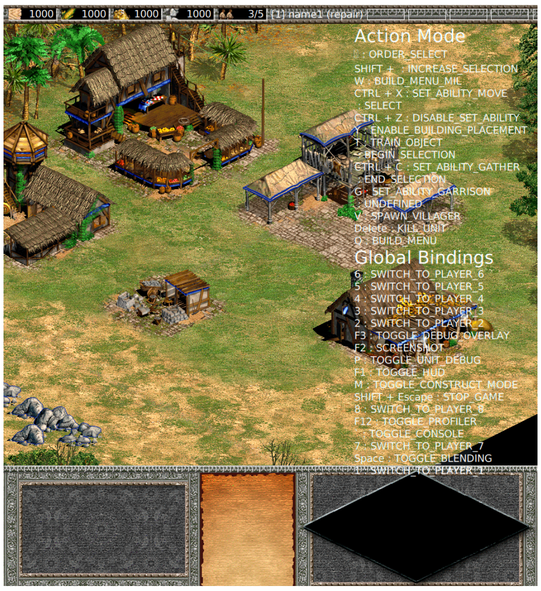


Figure 3.3: OpenAge Screenshot

4 Event-based Game Engine Concept

Within this work the game state evolving over time while playing a game is logically divided into section of a timeline containing the past and the future. The game state contains all values of all objects in the form of key-frames as well as the events that are executed in order to generate more key-frames. The past contains the key-frames that have already been calculated. The key-frames also store information about the full history of the game-state, the currently displayed frame and at least one future frame. The future contains events that generate more key-frames. Events are only executed at the time when their referenced time is relevant for the displaying logic, as they are going to change with a high probability, or might not happen at all. The game state can be advanced forward through the execution of events, filling the past section with more data to be displayed eventually. It is easy to reschedule or cancel an event it until it is executed, if the conditions change. For the past section, this work uses the curve logic, as described in chapter 4.1. The future logic is a event system that is optimized for the time-scheduled execution, as described in chapter 4.2. Whenever something in a curve changes, it is reacted to by the event logic. The method used to interlink these two technologies in order to monitor for changes is described in chapter 4.3.

4.1 Curves

Curves are values that are tracked over time using interpolation between key-frames. Defined curves are evaluated continuously within their defined ranges. A key-frame is a tuple of a specific time in milliseconds defining when this value is set, and the value. The type of value defines which type of interpolation is supported.

A curve is a description of how the value of within the gamestate changes with time. It is defined as a function $f(t) = v$, with v as the value of the described variable at time t. This method is an interpolation on top of key-frames, which are added as the game progresses. A key-frame is defined as the tuple $k := (t, v)$ with t as the time and v as the respective value. A specific key-frame is noted as $k_i = (t_i, v_i)$.

Every curve is constructed using a default-constructed value at time 0. This ensures that a curve is always defined and can it is always possible to evaluate it at one point in time.

For higher order interpolation the value type has to support basic mathematics including multiplication with a constant and addition with itself 4.1.1. For discrete, step-function interpolation the value is any copy-constructible type, since it does not require any calculation to deduce the value. Discrete curves are described in section 4.1.2. There are also curve-containers, that can hold any number of objects. One container type can track the creation and destruction of contained objects in curved sets as described in 4.1.3. The second container type contains the impulse occurrence of values valid at one single point in time, which are curved queues as described in 4.1.4.

4.1.1 Continuous Curve

Continuous curves represent the world as a linear interpolation between set key-frames. It is used for any continuous change or movement, such as building progress or unit position. If the requested time lies outside the range of already defined key-frames, the last currently defined key-frame is assumed, with the interpolation staying constant. This means that a value with no further key-frames stays constant, as shown in figure 4.1.

For a continuous curve the *value* has to support multiplication with a constant, and addition and subtraction with the same type have to be defined. The continuous value is interpolated at time t using equation 4.1, with two key-frames k_i, k_j that surround the value at time t with v_i as the value of the key-frame k_i at time t_i.

$$continuous(t) = v_i + (v_i - v_j) \cdot \frac{t - t_i}{t_j - t_i} \tag{4.1}$$

with the key-frame times defined as the closest key-frames to reference time.

$$t_i = \max\{t_i | k_i = (t_i, v_i) \text{ with } t_i \leq t\}$$

$$t_j = \min\{t_j | k_j = (t_j, v_j) \text{ with } t_j > t\}$$

$$t_i \leq t < t_j$$

If either of t_i, t_j is undefined, the value $\max\{t_i|k\}$ or $\min\{t_j|k\}$ is assumed respectively.

Figure 4.1 shows an example for the progress of a building being built, and subsequently existing in its finished state. The vertical dotted line represents an evaluation using the two marked key-frames k_i and k_j. Previously calculated event output is shown on the left-hand side of the vertical dotted line, while the current prediction of the future is shown to its right-hand side.

Functions of higher order than linear interpolation can be closely approximated using short key-frame intervals to be usable in the game engine, as stated in [Smi13].

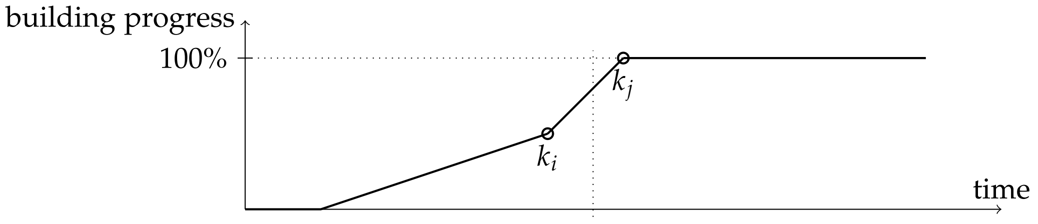


Figure 4.1: Continuous curve
A continuous curve of a building construction having one, later two workers.

4.1.2 Discrete Curve

It is also possible to use a step function for interpolation, which is defined as the discrete interpolation in equation 4.2, using v_i as the value of the key-frame k_i right before t.

$$discrete(t) = v_i \tag{4.2}$$

$$t_i = \max\{t_i | k_i(t_i, v_i) \text{ with } t_i < t\}$$

Values represented by a discrete curve do not need to fulfill any special type requirements. Therefore a discrete curve can hold any type, such as objects and strings. Discrete curves hold values that have logical reasons that they should not be interpolated, such as resources or population count which is always discretely defined. They can also hold values that do not support interpolation, for example references or pointers to other objects.

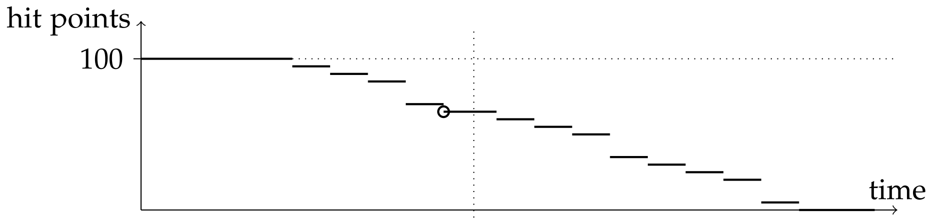


Figure 4.2: Discrete curve showing hit points of a building being attacked
The building initially possesses 100 hit points. The irregular vertical distances between the short plateaus are due to the building being hit with different amounts of damage. The vertical dotted line represents an exemplary evaluation of the curve, using the marked key-frame.

It is important to note that game-changing variables such as hit points are stored in discrete curves. These values are not designed for interpolation, since an interpolated, and therefore delayed, removal of a unit from the battlefield can change the future in major ways.

Figure 4.2 shows an example how the hit points of a building being attacked are reduced. The building is hit with different damage dealt, resulting in the non linear reduction of hit points.

4.1.3 Curved Sets

Curves are also used to describe containers such as sets. A set is iterated over a snapshot at a given time t, were all values that are marked as *alive* during that time frame will be listed. The listed set contains all objects whose time of birth happened before, and time of death will happen after the given time, as formalized in equation 4.3.

$$set(t) = \{o | o(t_{birth}) \leq t < o(t_{death})\} \tag{4.3}$$

t_{birth} is set to $+\infty$, if the object has no scheduled time of birth. t_{death} is set to $+\infty$, it the object has no scheduled time of death. Curved sets can contain other curved values, which track their own value over the livespan of the container. The developer has to make sure to set the death time of objects in the set because the set does not track hit point values and cannot detect deaths on its own.

Figure 4.3 shows an example of a set with different live spans of registered objects.

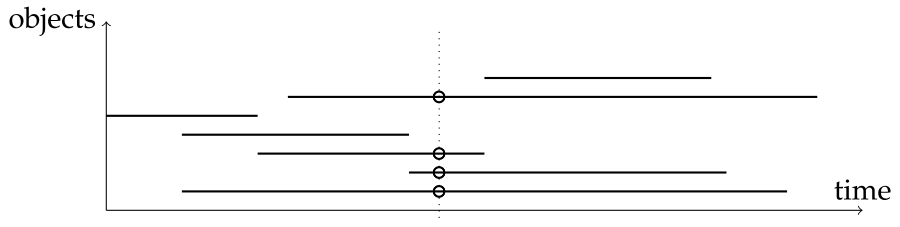


Figure 4.3: Curved Set
A curved set of alive units. There are two units that were created and destroyed before the evaluated time, visible starting and ending on the left-hand side of the vertical dotted line. The units that are alive at the given time are marked with circles.

4.1.4 Curved Queues

First-in-first-out (FiFo) queues work similar to pipelines, in which the first value inserted is first to be taken out again. FiFo-queues are also an integral component of the curved world, in which they represent building-queues, action-queues, and other lists.

Queues consist of singular, non-lasting events o at time o_t. Events are accessed by selecting all events that happen between the times t_{from} and t_{to}:

$$queue(t_{from}, t_{to}) = \{o_t | t_{from} \leq o_t < t_{to}\} \tag{4.4}$$

In a curved queue an object has exactly one time when it happens. Curved queues are used to describe pending objects, that will happen within the requested time frame. In figure 4.4 several building events from a factory are shown.

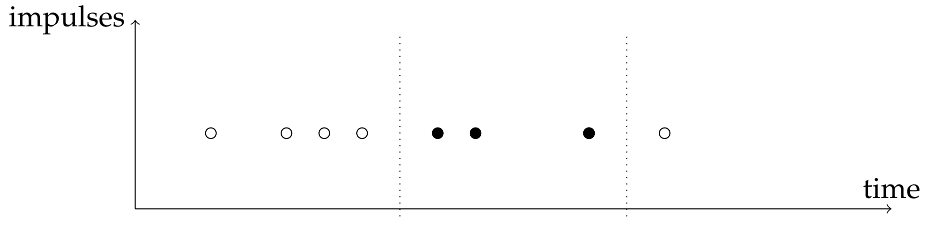


Figure 4.4: Curved Queue
A curved queue with events happening at the indicated times. The two vertical lines indicate the from- and the to-marker. The three filled circles represent the events that are active within the defined time-frame between the dotted lines, while the empty circles show events that happen outside of the chosen time-frame.

4.2 Events

Event driven systems engineering has been solved for many programming languages [Fai], therefore this chapter will focus on the differences compared to a standard event-system. The event system does not provide a notification logic, as is used commonly. It allows developers to create events triggered at pre-calculated times. These events are generated in response to either user input or the predicted fulfillment of conditions such as the completion of a building. For example a user input event, such as a change of destination for an already moving unit, can lead to the internal rescheduling of already planned events. A change within an event can either lead to the event happening at a different time, earlier or later, or not at all. The event "reset game" may happen earlier, if the player's panel misses and the speed of the ball is increased. The same event may happen later, if the ball is slowed down, and it does not happen at all if the player moves and hits the panel.

Traditional event systems provide functionalities to trigger events when predefined conditions apply or change. With the proposed event system the time when a condition is met, and therefore an event is triggered, is predicted. For example, in order to monitor change, it is mandatory to not only execute events at pre-calculated times. Predictions need to change whenever one of the input parameters changes, and need to recalculate according to the changed conditions.

The execution of events is triggered as a to request to execute all events up until a certain time. The game logic engine is iterating over the events that have a associated time *t* before the requested time. The order of events in the time line is kept in order to execute earlier events first. This execution is triggered regularly for example by the graphics. Because these events are usually less frequent than graphics frames, the execution for an event can take longer than one frame. An event can generate data for multiple frames using curves by predicting a key-frame far into the future, A prediction is not re-run until a change in preconditions has happened.

While creating an exemplary game logic it was necessary to create the following event types in order to generate hook points for different functionalities.

once(time)	Is triggered at the given time and is not repeated.
on-execute(time)	Is triggered when the execution is about to execute this point in time.
on-change(time)	Is triggered when the relevant timeframe comes into play, and the value has changed to the state when the trigger was registered.

on-change-immediately	Is triggered upon a change of something somewhere in the timeline.
on-pass-keyframe	Is triggered when a key-frame is not relevant for the interpolation anymore.

Some events are recalculated, whenever a dependency changes. As conditions can change before an event is executed it is important to recalculate the prediction. The following chapters describe the above-mentioned types of events and their interactions with the curve environment.

Every event prediction consists of the time of execution, the callback, and the single target it is called on. Some events also have a list of dependencies that are monitored for changes and can trigger the re-calculation of the prediction.

The target for the event is either an object from the game state, such as the ball, or any value from within, such as position and speed. The target is also passed to the callback, in order to reuse one event implementation for multiple events. For example to move a players panel, an event "player.move_panel" is defined and instantiated twice, once for each of the panels of player A and player B.

4.2.1 Event Type `once`

The `once(time)` event type is called at the time defined to issue the event. The event defines monitored dependencies, in order to recalculate the exact point of execution. If the event is canceled during recalculation, it is removed and no effort is made to revive it.

This event type is used for user interaction, as it usually only requires one single response, with no predictable repetitions. It is applied for the completion of research and other game-internal events that happen only once, or if cyclic behaviour is unpredictable.

4.2.2 Event Type `on-execute`

The `on-execute(time)` event type is called at the defined time and it is re-predicted and repeated after it is executed. If a prediction disables the event, it is not entirely removed. It is kept on hold, and is recalculated whenever one of the dependencies changes. This mechanism allows the event to re-enable itself in order to come back into action when the conditions are met again.

This event is used for cyclically appearing events that are possible to predict. It is applied for cyclic events such as damage dealt by a defence tower guarding an area within a defined range. The dependency list of the "ball.reflect_wall" is dependent on the speed of the ball, because it is re-predicted, if the speed changes. A wall reflection is also rescheduled as soon as it was executed.

4.2.3 Event Type `on-change`

The `on-change(time)` event type is triggered at the defined time whenever a dependency value changes. The exact execution time is determined when the change happens. This event type is re-activated after it has been executed.

This event type is only executed if the targeted value is changed between the time of registration and the planned time of execution. This is required for example for path planning, to “protect” the calculated path. If a obstacle, such as another ball, moves into the path of the ball, the path has to be changed. No re-prediction is necessary, if no such influence occurs.

4.2.4 Event Type `on-change-immediately`

The `on-change-immediately` event type behaves similar to the `on-change` event type, but instead of triggering at the time given on activation, it is triggered immediately after a change has happened. This decouples the execution of this event type completely from the normal timeline, and provides a method of reacting to changes immediately, and not only on changes in the future.

This event is not used during normal game play, because it disrupts the execution workflow, and can also use a lot of resources, if the event needs recalculation for every change that happens. It is preferable to use the `on-change` type, as it accumulates changes until the event is executed, and only executes once for any number of changes that have happened before.

4.2.5 Event Type `on-keyframe`

The `on-keyframe` event type is used for managing the internal integrity of the curve constructs. Whenever a key-frame has passed the scope, all previous key-frames can be erased from memory in order to free up memory. The scope is defined by the *now*-time, which is the time that is currently displayed on-screen. If a key-frame is not involved in the interpolation for future values, this event is triggered, receiving the associated time of the key-frames.

The event is ignored, however, if the memory consumption of the curve is unlimited, it may eventually use all main memory. It is therefore the most effective solution to delete the past key-frames, freeing the consumed memory. Another solution to free memory, writing it all to disk, is discussed in 4.4.

4.2.6 Example

The correlation between curves and events is best shown using a small and self-contained example. Within the example, the different event types are applied to a continuous curve and a change is made. The curve, as displayed in figure 4.5, and the events that are associated according to table 4.1, are executed. The continuous curve through key-frames k_0 and k_1 is modified at the time t_0 for the time t_2:

```
curve.set_drop(t2, 0); // will remove k1 at t1
curve.set_insert(t3, 1);
```

The key-frame k_1 is deleted and replaced with $k_2 := (t_2, 0)$ and $k_3 := (t_3, 1)$.

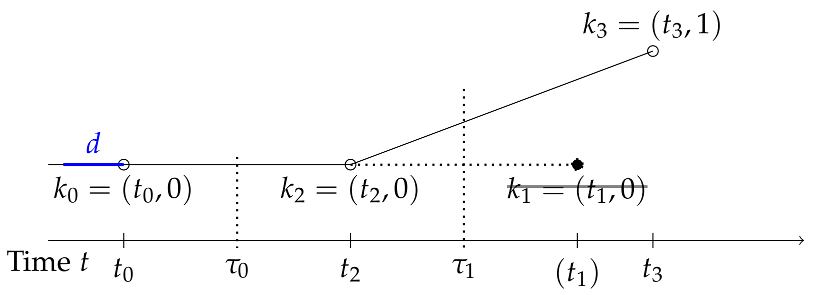


Figure 4.5: Visualisation of a change in the timeline

event type	parameter	triggered at
`on-change-immediately()`	–	t_0
`on-execute(time)`	t_0	$t_0 - d$
	τ_0	$\tau_0 - d$
	τ_1	$\tau_1 - d$
	t_1	$t_1 - d$, without k_1
	t_2	$t_2 - d$
	t_3	$t_3 - d$
`on-change(time)`	t_0	never
	τ_0	never
	τ_1	$\tau_1 - d$
	t_1	$t_2 - d$
`on-pass-keyframe()`	–	t_2 with k_0, t_3 with k_2

Table 4.1: Trigger calling times

Table 4.1 describes which trigger is called and when, depending on the arguments and the values of the curve.

d represents the time period between the last frame and the next frame, as events are executed regularly and in batches, as described in 4.2.7. All events are executed within this time period, for example for a next frame at $\tau_0 i$, all events planned to happen at times $t < \tau_0$ are executed. According to the mapping in table 4.1 this includes the events `on-change-immediately(`t_0`)` and `on-execute(`t_0`)`.

The most simple event type is `on-change-immediately`. It is triggered exactly when the change happens at t_0. It is registered without any parameter.

The `on-execute` event is called in the time-frame before the time in its parameter is relevant for the rendering. If it is registered at t_0, it is called in the time range d before t_0 has to be ready. The value of the curve at t_0, τ_0, t_2 is 0. τ_1, t_3 will evaluate to the linear interpolation on the curve between k_2 and k_3. t_1 is not bound to a key-frame anymore and is interpolated between k_2 and k_3.

The `on-pass-keyframe` is called at times when the next key-frame has been passed, and the previous one can safely be dropped.

The `on-change` event is called if the value the event was pointed to was changed since the event was registered. The value between k_0 and k_2 was not changed, therefore any registered change events between k_0 and k_2 is not triggered. However, at the points where the value was changed, the event behaves in the same manner as `on-execute` and calls d before the time is relevant for the execution.

4.2.7 Execution of Events

In the main game loop the event execution is triggered to create the next frame. The time of the frame is handed over to the event execution to execute the relevant events. This time is set in a way, that even more complex calculations can finish in time before the result of the execution gets relevant for the next requested frame. If in a subsequent frame an event arrives, which was required for a previous frame, it is still executed, but all events happening afterwards are recalculated. This is important, because the conditions may have changed, and the result of a later event may have been overwritten.

With the set time a subset of events are extracted from the event queue, if they are scheduled to happen before this time. These events are then executed sequentially as a batch job of the queue.

User input may generate events that are scheduled before frames, that have already been executed. If such an event is received, all events that were scheduled after the new event are executed again. The time, that such an event is applied in the "past" is limited, to avoid heavy recalculation chains and a possible loophole for cheater.

The set time of the execution of events might be a few frames in the future. The number of frames this execution is scheduled before the time it is actually drawn on screen depends on the scenario. In a game with a low frequency of user input, it is possible to calculate multiple frames in advance, because it is not likely that a user produces events that are scheduled before the ones already executed. In a game with a high frequency of user input additional events that are scheduled at a time before the ones that have already been executed requires more frequent recalculation. This resulting in a higher computing load than necessary.

The execution queue is not modified during an execution cycle by automatically generated change events. Change events are stored sorted in a background buffer. This change buffer is converted into events, that are executed after current queue is completed. The detailed change logic is described in chapter 4.3. Normal, manually created events are immediately inserted into the execution queue in order to avoid re-calculation.

4.2.8 Event Rescheduling

An event is re-predicted if a change in a dependency happened. All changes during a batch execution of the event queue are accumulated. Events with a changed dependency are then re-predicted at the end of a batch execution. Re-prediction are executed frequently, because every change might have large consequences on a single event. An event can decide, that it is not scheduled again during the re-scheduling process. It does this by returning $+\infty$ from the prediction function. The event is removed or only not executed, depending on the event type. The event is removed completely from the execution if it is not designed for reactivation or is paused from execution and can be rescheduled later, if it is a reoccurring event.

4.3 Changes

When a value of a curve changes the `on-change`-events is triggered as well as all events, that depend on the changed value. A change can happen for example by interaction of the player or by the continued execution of the normal game flow.

A change and its execution can lead into deadlocks and dependency loops, that is why it is important to address the special handling of those changes.

In the naive implementation, a change-callback is called in the setter-function of the curve. The changing method would have to wait, until the change callback has completed. If the change callback now changes another value, another callback is called and so on, leading to a recursive deadlock of callbacks, and furthermore a crash of the running program.

So to address the issue of recursive deadlocks, the changes is tracked and executed out of bounds. That means to track the changes a callback has made and execute them after the changing callback has completed. This decouples the change-callbacks from the changing logic itself and avoids the recursion. This approach itself is also error-prone, if the event system itself is not acyclic.

If an event changes a value, the change callback is inserted into the change-queue. The change queue is then executed when the callback has completed, possibly changing the same value again. This on the other hand would lead to the same callback being inserted into the change-queue again, leading to a cyclic dependency within the logic.

One solution is, to detect such a cyclic dependency during run time using the methods as described in [Jon+08] and [CCG02]. But these approaches are infeasible, since the flexibility is given to the developers, that they can design however the engine should behave. Also, it is required to change a curve twice, in order to insert one bend, as seen in the example 4.5, which would require dependency tracking.

The method applied here is the use of a double buffer and change accumulation.

In the execution phase all events that is triggered are executed in the time sequence they appeared. When one of these events changes the key-frames of a curve it is checked, if any event depends on this change. If it does, it is stored into a pool for later.

This pool is used to eliminate duplicates - so if one value was changed multiple times during the execution phase it is still only stored one time. It is implemented as a Hash-Map that uses the eventclass and the target id for its hash. The earlieset time for multiple changes is selected, because every later change is handled by this one call to the event-callback. If a change has already been processed within this frame, it is not added again. Instead, it is pushed into the next iteration of event executions. This is done using double-buffering between the current and the next frame. In the beginning of the next execution the buffers are swapped.

In the change application phase the changes are then processed and re-added into the event queue. This is then repeated until the change and the event queue are empty. Possible deadlock situations are solved, as described above, using double buffering and backing off execution into the next frame.

4.4 Serialization

For multiplayer games it is important enable multiple devices to share the same game state. To enable the transmission, it is required to serialize the running curves into a stream of changes and updates, and apply it to a running game state. Every curve is serialized individually by serializing the operation, the key-frame time, and the associated data. To reconstruct the curve, each curve needs a unique identifier.

Listing 4.1 provides an exemplary serialization of the position of a ball in the *Pong* game into a *json* interface. The ball's position caries the identifier "1", its speed the identifier "2". In this example it is important to note that the key-frames are sorted by the times they are defined. This procedure is important for applying the key-frames back into a running game-state on the receivers end. Speed and position are serialized alternatingly, because in *Pong*-scenario the key-frames for both speed and position are always set at the same time.

Listing 4.1: Serialized key-frame

```
[{'time':102,'id':1,'op':'set_drop','value':[102,0]},
 {'time':102,'id':2,'op':'set_drop','value':[-1,-1]},
 {'time':152,'id':1,'op':'set_insert','value':[40,56]},
 {'time':152,'id':2,'op':'set_insert','value':[-1,1]},
 {'time':205,'id':1,'op':'set_insert','value':[0,23]},
 {'time':205,'id':2,'op':'set_insert','value':[-1,1]},]
```

The exact definition and procedure on how the networking works on top of the curve is not in the focus of this thesis.

Serialization is also important for storing key-frames to disk. The disk-serialization is registering the `on-pass-keyframe` event for notification when a key-frame is in the past. The key-frame is subsequently serialized in the same way as for networking. This list of key-frames is used as a save-game, as well as the replay of the game played up to this point.

5 Implementation

In the scope of this thesis the game engine has been developed and applied to a simplified subset of the game, which is a game similar to *Pong*, in order to demonstrate the features of the engine. In *Pong* two players move their paddles up and down the left and right edge of the screen, while trying to play a ball between them. The concept is similar to the concept of air-hockey. The ball reflects from the top and bottom edge of the screen and from the players' paddles. The players' lives count down from three, removing one life every time the player's panel misses the ball.

The described game implementation at is base provides the same challenges to the game logic engine as exist for real-time strategy games. Path-following is a large component in both types of games, as is user interaction. Independent objects in both game types behave according to their state, which may change over time.

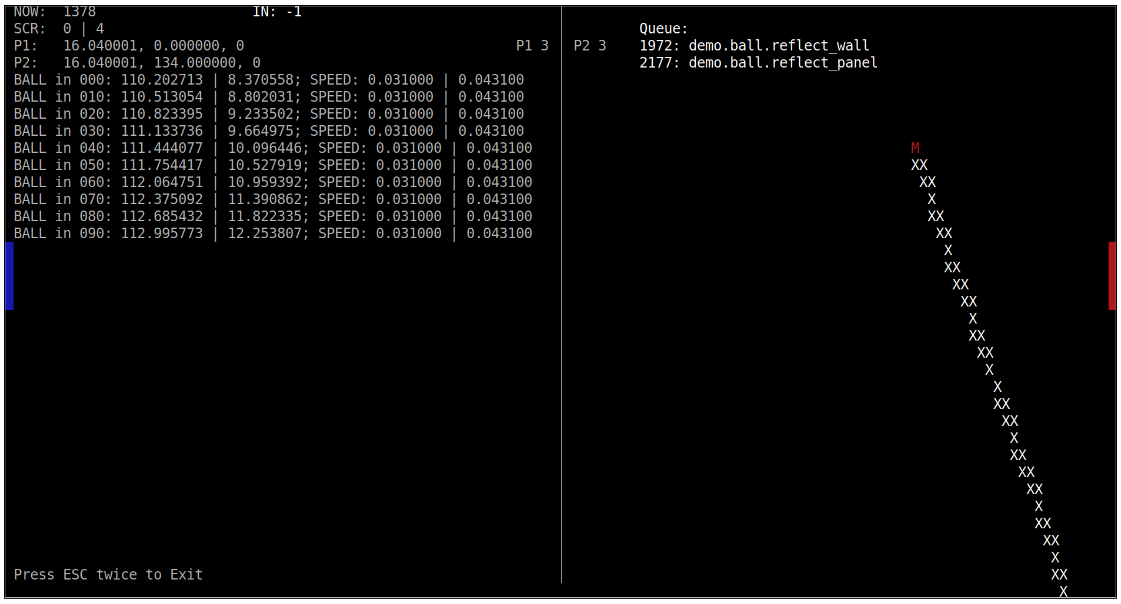
The simplified approach taken from *Pong* was therefore tested for its applicability within real-time strategy games such as *openage*, which is an open-source real-time strategy game based on the game *Age of Empires*, as described in 5.2.

This chapter follows the researched solutions and describes why and how the specific design decisions were made. The game-state is described using curves in chapter 5.1.1, including an exemplary implementation. The simplified version of the event system, called triggers, is described in chapter 5.1.2, along with their application to the event system itself. The event system and its application is described in chapter 5.1.3

5.1 Proof of Concept

A proof of concept was designed with the goal to keep the challenges minimal and solvable and still come up with as many challenges as they are in real-time strategy games. It was decided to use a a *Pong* clone using text-based rendering in order to demonstrate these features. Simple text-based rendering utilizing *ncurses* was found appropriate in order to keep the proof of concept simple and self contained, yet complete. *ncurses* is a terminal-based text presenter allowing a developer to draw any character at any position within a terminal in curses mode.

The a screenshot of the running game is in figure 5.1. The current time of the screenshot was at millisecond 1378 of the running game. The red and blue paddle of the players are located on the left and right hand side of the screen. The ball, moving to the bottom right, is represented by the red "M" in the right half of the screen. White "X" mark the path of the ball, as far as it is currently predicted. On the top left corner of the screen different variables are printed. These are from top to bottom: the current "now" value, player states, position of player 1 and player 2, and the prediction of the ball in steps of 10 millisecond. On the top right hand corner of the screen the event queue with two events is displayed The event `demo.ball.reflect_wall` is scheduled for executed at 1972 millisecond. The event `demo.ball.reflect_panel` is scheduled for execution at millisecond 2177, which will then check if the player's paddle is in a position to reflect.



```
NOW:  1378                   IN: -1
SCR:  0 | 4                                                                          Queue:
P1:   16.040001, 0.000000, 0                                 P1 3  |  P2 3    1972: demo.ball.reflect_wall
P2:   16.040001, 134.000000, 0                                                2177: demo.ball.reflect_panel
BALL in 000: 110.202713 | 8.370558; SPEED: 0.031000 | 0.043100
BALL in 010: 110.513054 | 8.802031; SPEED: 0.031000 | 0.043100
BALL in 020: 110.823395 | 9.233502; SPEED: 0.031000 | 0.043100
BALL in 030: 111.133736 | 9.664975; SPEED: 0.031000 | 0.043100
BALL in 040: 111.444077 | 10.096446; SPEED: 0.031000 | 0.043100
BALL in 050: 111.754417 | 10.527919; SPEED: 0.031000 | 0.043100
BALL in 060: 112.064751 | 10.959392; SPEED: 0.031000 | 0.043100
BALL in 070: 112.375092 | 11.390862; SPEED: 0.031000 | 0.043100
BALL in 080: 112.685432 | 11.822335; SPEED: 0.031000 | 0.043100
BALL in 090: 112.995773 | 12.253807; SPEED: 0.031000 | 0.043100

Press ESC twice to Exit
```

Figure 5.1: A running game of *Pong*
The ball is marked as the red "M", moving to the bottom right as indicated by the predicted positions marked with "X".

5.1.1 Game-State: Curves

A simple game state was designed for the proof of concept consisting of the two players and one ball as described in chapter 5.1. The different values within the state are created using discrete and continuous curves. More complex objects within the game state are implemented using the `EventTarget` interface, to group all properties for into one group for events to target. The game state in listing 5.1 is sufficient for the *Pong*-game to run.

Listing 5.1: *Pong* game state

```
class PongPlayer : public EventTarget {
public:
	curve::Discrete<float> speed;
	curve::Continuous<float> position;
	curve::Discrete<int> lives;
//...
};
class PongBall : public EventTarget {
public:
	curve::Discrete<Vector<2>> speed;
	curve::Continuous<Vector<2>> position;
};
class State {
public:
	PongPlayer p1;
	PongPlayer p2;
	PongBall ball;
};
```

with the root of the game state being the class `State` containing two `PongPlayers` and the `PongBall`.

The `PongPlayer` contains a discrete speed value because speed is interpolated, as it is simply reflected every time it collides with a wall or a paddle. Lives are also discrete from the inherent logic. The position of each player's paddle is a continuous curve, because it is interpolated between the key-frames.

Both player and ball are event-targets to function as a dependency and targets for the event system. This topic is discussed further in section 5.1.3.

5.1.2 Triggers

Curves themselves do not offer any logic for interaction design. So-called triggers are used to design interactions for the curves. Triggers are designed to modify curves at a defined time, if defined conditions apply. A trigger is a callback defined with a time on a certain curve. The callback does not perform any re-predictions or change monitoring, as provided by the higher level event system. Upon registration, a trigger is inserted into the execution queue. The execution queue is sorted by the time its events are to scheduled for execution. Upon execution of a frame in the queue this list are iterated, and the events between the last frame-time and the defined frame-time are executed sequentially. As described in section 4.2.7, the time is defined by the next rendered frames.

Listing 5.2: Ball reflection using the trigger interface

```cpp
// time: time when the ball hits the other side of the field
void reflect_wall(PongState &state, const curve::curve_time_t &now) {
  auto pos = state.ball.position.get(now);
  auto speed = state.ball.speed.get(now);
  speed[1] *= -1.0;
  state.ball.speed.set_drop(now, speed);
  state.ball.position.set_drop(now, pos);
  predict_reflect_wall(state, queue, now);
}
```

Listing 5.2 shows how a function is registered at the trigger interface using plain function pointers. The function `reflect_wall` will trigger reflections of the ball on the top or bottom of the screen, turning the movement speed of the ball around and triggering the re-prediction of the event.

Listing 5.3: Prediction of the time when the ball is reflected at the wall

```cpp
void predict_reflect_panel(PongState &state,
                           const curve::curve_time_t &now) {
  auto pos = state.ball.position.get(now);
  auto speed = state.ball.speed.get(now);
  double time_delta = 0;
  // ... calculate time_delta as the time until the next wall
  auto hit_pos = pos + speed * time_delta;
  state.ball.position.set_drop(now + time_delta, hit_pos);
  TriggerManager.on(now + time_delta,
    [](PongState &state, const curve::curve_time_t &now) {
      reflect_wall(state, now);
  });
```

The prediction logic is shown in listing 5.3. First, position and speed of the ball are extracted from the curves. Second, the time `time_delta` until the next hit on a wall is calculated, depending on the movement direction. Third, this new time is used to set the exact point, when the ball collides with the wall using the speed. Fourth, the event defined above is re-inserted into the queue. The defined time of the re-inserted event is set in the future, because the ball has to cross the field again first.

There are two types of triggers: one for reflection from the walls (reflection in y-direction), and one for reflection from the panels (reflection in x-direction). Because these axes are orthogonal, the triggers for the axis are executed independently.

The low-level trigger logic is sufficient for defining simple and predictable dependencies. The trigger logic is not sufficient if an event has to continuously check for its condition and tune the exact point of execution. This is only possible by monitoring dependencies and providing the functionalities to recalculate. For example, if the *Pong* game-play is extended with random reflections, all triggers are manually re-calculated, leading to a highly coupled interface. A higher-level interface alongside the trigger logic is the event interface, which is especially designed handle the dependency logic.

5.1.3 Events

On top of the trigger interface it is required that events need to contain more functionality, especially re-prediction and dependency management. The logic of *Pong*, together with the planned application in a real-time strategy game, requires the use of more than one event type, as it is the case using triggers. These event types are explained in table 5.1.

These event types are managed with `EventClasses`, which define the actions to take in order to re-predict the execution time. Also, every object within the game state has to implement the interface for `EventTarget`.

An `Event` is derived from applying the `EventClass` to the `EventTarget`, maybe accepting additional arguments. This is controlled by the `EventManager`, which keeps track of all registered event classes, the queue, and the changes.

An event is registered at a defined time as follows:

Listing 5.4: Event registration

```
eventmanager.on(unit.damage,
                target,
                GameState,
                time,
                {{damage, 1}});
```

This applies the event "`unit.damage`" to the `target`, which is an `EventTarget`, using the `GameState`, the `time` and the additional parameter "damage". The `GameState` argument provides the full game-state. The event is scheduled for the time `time`, with the additional parameter "damage" defining the amount of damage.

`ONCE`	Is triggered exactly once, but is rescheduled whenever a dependency changes. This is used for player interaction events such as movement.
`ON_EXECUTE`	Is triggered at the set time without further conditions necessary. The initial time is calculated using the re-prediction mechanics. When the event has been executed, it is rescheduled immediately without the need to trigger it again. This is used for regulary game logic, such as wall reflection.
`ON_CHANGE`	Is executed at the set time, if the target value has changed. When a dependency changes, the execution time is reevaluated. : This is used to observe whether a decision that is still in the future stays valid.
`ON_CHANGE _IMMEDIATELY`	Is executed immediately after a change was made, independently of the set time. It has no option for rescheduling, because it has no planned time of execution. This is used to make general assumptions over the gamestate, that do not depend on the actual time the change comes into execution.
`ON_KEYFRAME`	Is executed whenever a curve has passed a key-frame, and this key-frame is now in the past. This is only useful for internal house-keeping, for example forgetting the history or writing it to disk to save memory.

Table 5.1: Implemented event types

5.2 Application in Real-Time Strategy Games

Based on the implementation done for a *Pong*-clone it is possible to create a game state for a real-time strategy game. At the core of the game architecture lays the game state defined by curves. A snapshot of a curve is made in order to interface with the different parts of the game engine, such as graphics, sound, and AI. Snapshots contain the key-frames that are relevant for the current time, reducing the amount of data required for the snapshot while providing the complete world.

Objects within the game state consist of properties, abilities, and attributes. Properties are information shared across all units and required for basic functionality such as hit-points and position. Properties are implemented as curves in order to enable the tracking of these values. Abilities are actions an object may take, such as movement, attacking and producing of units. Abilities are implemented as `EventClasses`, because when an action is triggered either by the user or by the game mechanics, an `Event` is derived from the ability. The application and features of `EventClasses` is described in chapter 5.2.1 Attributes, as described in chapter 5.2.2, define which types of actions can target an object. The player themselfes possesses some attributes, such as their resources and population limit.

5.2.1 `EventClass`

An `EventClass` is defined by the event type, as described in chapter 5.1.3, the callback to trigger, the recalculation method, and its setup routine. `EventClasses` are registered at the game logic engine, for calls using their descriptive name as seen in listing 5.4. This name enables the possibility to decouple the different events while maintaining a high quality debug output. An instantiated `Event` consists of the tuple `EventClass` and the target `EventTarget`. `EventTargets` are special classes for example `PongPlayer` and `PongBall` in listing 5.1. Curve types are also `EventTargets`. The `EventTarget` provides functionality to track the values contained within itself.

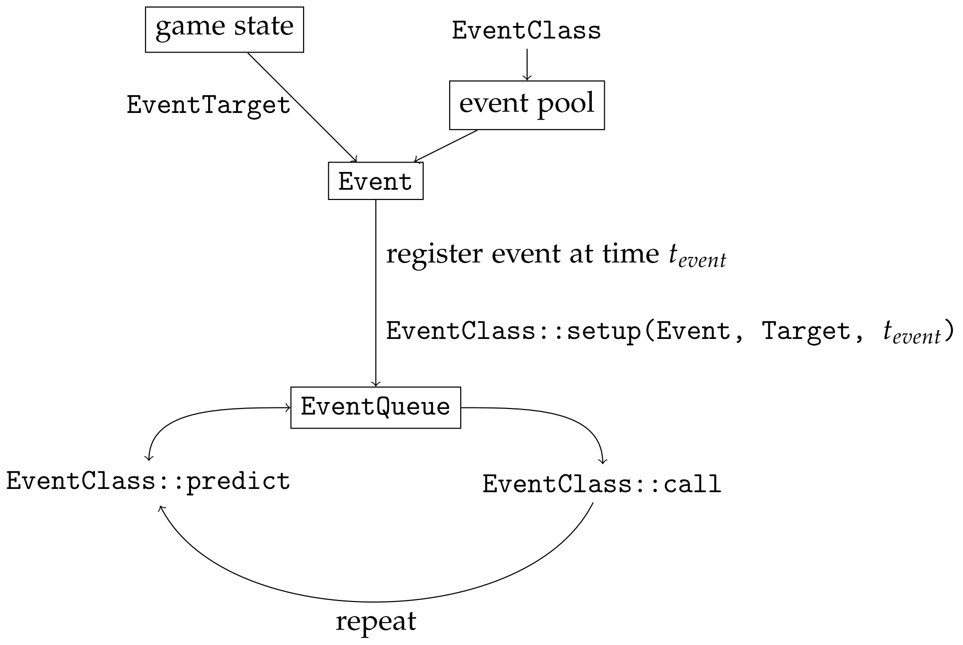


Figure 5.2: `EventClass` workflow

Figure 5.2 describes the flow control of the `EventClass`. First an `EventClass` is registered at the event pool, being accessible by the defined full-string identifier. The game state consists of `EventTarget`, that hold the values of the currently running game in form of curves. An `Event` is created by calling an `EventClass` on a certain `EventTarget` to create an event invocation with a reference time t. This instance also contains the additional parameters that have been specified on invocation. This tuple consisting of the event instance, the target and the time is passed on to the `EventClass::setup` method, that will register all dependencies within the target and the remaining game-state. After the event is created and its dependencies are defined, it is stored in the `EventQueue` to wait for its execution. The actual invocation, the time t_{event} of the event can be rescheduled by the `predict` method whenever a registered dependency is changed. This method can reschedule the event, it can remove it from execution or reorder the event execution pipeline. When the `EventQueue` is requested to execute all events up until a time $t_{now} > t_{event}$, the event classes *call* method will be called, with the target, the time, and the event instance containing the additional parameters.

Certain event classes are being rescheduled after execution, which is done by calling the `predict` method again, with t_{event} as the reference time for calculating the next occurrence.

5.2.2 Attributes and Actions

Attributes are either shared, in a way that the existence of the attribute is defined by the units objects type, or they are unshared, which defines attributes only a single unit holds based on its actions.

Shared attributes are information stored for this unit type. It changes over time, however this is an issue for the game designer utilising this concept. Shared attributes contain information that does not change as often as for example the health points of a unit.

These can be: *armor, attack, heal, hitpoints, population, speed*

Unshared attributes are unique for each unit. These attributes are used to track and the individual behaviour of a single unit. These values usually change more frequently than shared attributes.

These are: *direction, formation, garrisson, projectile, resource*

If an action is applicable to another unit is dependent on the units abilities. For example an object that does not have the *resource*-ability can not be targeted by a *GatherAction*.

Actions are the actions a game state object can perform. They are heavily dependent on the actual object. For example a production building has a *TrainAction* which can produce new units, while a *Worker* object can create new build-sites and work on those build-sites.

Actions are generally implemented as `EventClasses`, as they are invoked by a command, accepting a single target, with little to no dependencies within the full game-state.

Using this separation into actions, that can target objects that have certain abilities it is possible to implement the majority of interaction logic within the real-time strategy game.

6 Discussion

An event-driven logic is an uncommon approach in game logic engines although it is widely spread in other areas, such as systems engineering or web development. Using techniques borrowed from local event-based mechanisms such as python asyncio, the concept itself is known to developers. Using small, self-contained events as descriptions for interaction mechanics enables developers to decouple the different parts of the logic using the same core event mechanism. If for example a user input is received, it is translated into an event of type `once`. The same input, however, is received from a network component if it is transmitted to the game server. By applying the same arguments and state to the same event, the generated event will have the same end results.

Using curves to define the path of an object in the game world, as well as for any other changing value, has been proven beneficial within the scope of this thesis. Especially the possibility of decoupling the physics calculation from the graphics frame interpolation has been found advantageous. In *Pong*, as well as in most real time strategy games, the path a unit will take is predicted, and is represented using key-frames. This path does not need to be re-calculated for every frame and the unit's arrival time and location on the path is always clear. The path is only changed if an obstacle in the path changes the timing of the path itself.

Variable interconnection is an important feature for both, multi-player real time strategy games and the *Pong* game that uses curves. By applying the methods described in 4, it is possible to use the engine concept as it was shown in the proof of concept. In order to apply the concept to a real-time strategy game, an excessive pool of standard interaction functionalities, such as damaging of units, is required. The proof of concept has shown that such standard functions can are incorporated in the engine using their fully qualified names. The wall-reflection functionality is such a library function. It is always registered and makes the ball behave in the desired way, although extensions to the ball mechanics can change the default behaviour. The introduction of variable reflection patterns at the panels is a modification of the panel reflection behaviour, while all other logic still applies. This is possible through the use of a dependency list that includes the ball's position and speed. Similar situations are found in real-time strategy games, here the default behaviour of units is configured using context-aware events such as `on-change` and `on-execute` to formalize their actions.

An important feature for multiplayer-enabled game engines is the ability to effectively serialize their game state. Although this thesis does not compose a networking architecture to use with curves, the general concept to serialize a game state into a stream is discussed in chapter 4.4. Since the serialized game state is valid for multiple frames, it is possible to use only changes on the curves within this game state to synchronize over the network. Lock step based games often use a hybrid between input synchronization and game world synchronization by transmitting all input, and a part of the world, with each frame.

A real-time strategy game engine has to solve computational expensive tasks during its execution without losing to the graphics frame rate. Such a task is for example to find a long-distance path for a group of units. There is the approach of using a separate thread for the execution of such a task, but this still leaves the unit immobile until the full path has been calculated and it is committed back into the game state. Using a poll-event based approach, the unit requests a new path whenever it has completed the previously followed section, spreading the execution over multiple frames. This is also a possible approach in sequential game engines, but its high level of coupling deeply in the core of the game engine is producing large amounts of interconnected code. The benefit of the event-based approach is the improvement in readability and the increased decoupling of the subsystems in the game engine.

This game logic engine approach is advantageous when its logic is applied to a fully featured *Pong* game. It was also shown that real-time strategy games have largely similar demands to the game logic engine as the ones solved in the *Pong* engine. The event driven game engine using curves is favourable to a newly designed game, but it is challenging to implement it into already existing games using a lock step approach. The full game logic would need to convert into the new architecture. The benefits include a decoupled event interface, with event classes defined at startup and instantiated within the normal game workflow. Furthermore, it is a way to define a continuous game state that tracks its changes and allows the modification of values in the past and future.

The game engine is usable if the following preconditions are met:

- Command-based
- Few actions per minute
- Predictable game logic
- Game logic able to approximate using linear functions

Command-based means that the input to the game is not usually entered as direct control over a unit, but as commands to one or more units. The units themselves act independently once a command is received.

Few actions per minute means that the game engine is not feasible if the behaviour of a single unit is designed for change on a high frequency. Highly frequent input is the case in games in which the player controls one unit using direct keyboard input to move around, such as first-person shooters. Such a behaviour is not predictable and challenging to integrate into the event driven game engine on a large scale.

Predictable game logic is required, because curves are defined into the future. A high degree of randomness within the observed curves produces one key-frame per random movement, which will lead to one key-frame for every graphics frame. It therefore does not provide any benefits over exiting methods. Such a hard-to-predict scenario is better approached using finite-element-methods or lock-step methods.

Linear functions are used for interpolation in the curve engine, which means that for a game using curves the game is approximated using these linear functions.

7 Conclusion

This thesis evaluates a different approach for an event driven system, as it is used over prolonged periods of time. It was shown that it is possible to create an event system to trigger events at specific times. These times are introduced from the user, for example the duration of research the user requested as well as by interactions within the system itself such as one unit damaging another. The event system requires a game state representation that is valid at any requested time. If an event is executed, it needs the currently known game state at the time of execution. If there were no user input, events are executed as defined by the game logic. The Graphics subsystem requests the next frame-time. All events, that are registered before the frame time are subsequently executed.

The game-state is represented continuously in order to generate the game-state at the time of the event. This is achieved using curves as described in 4.1.

It was shown, that a game state is represented continuously using continuous and discrete curves and curved sets and queues. The biggest change in game logic is that the curve logic is used for every value in the game-state, since the event logic requires to track every value, that can change during the period of a game. It was shown, how events are used to describe the interactions between objects and their values and how the timely-triggered events can implement things such as *on-change* logic.

This concept and subsequently the API was used to redesign the game state of *openage*. Together with feedback from the community it was possible to provide a developer friendly interface.

Altogether it is shown that event driven game logic is not only usable design, but it provides a general and extendable and to define logical interactions and constraints in an environment designed exactly for the requirements of real time strategy games.

Further work on this game engine concept includes the networking and distribution logic in order to synchronize multiple game-states. This can be either achieved with incremental serialization of the changes, the events have locally applied by transmitting only the changes to every connected client. Or it can be achieved by transmitting all events to one central authority which will distribute the changed game-state.

Another important extension is to extend curves with the possibility to apply relative changes. This will lead to the ability, that a value has not to be exactly set at a time, but merely relative, which will be converted into an absolute value at the time of the event.

This will enable a optimized chaining of events, that not every change that is made in the future has to be guarded with its own event.

The third important extension to the curve and event logic is advanced debugging serialization, in which a developer can easily see all planned events and interactions. Such an interface is already proposed within the proof-of-concept, but it is currently only usable for positional values, which can be tracked as paths on the map.

List of Figures

List of Tables

List of Listings

Bibliography

[Aut09] Autodesk Maya Press. *Learning Autodesk Maya 2010*. 1st ed. John Wiley & Sons, 2009.

[CCG02] J. Cleland-Huang, C. K. Chang, and Y. Ge. "Supporting event based traceability through high-level recognition of change events." In: *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. IEEE. 2002, pp. 595–600.

[Eme13] E. Emerson. "Crowd Pathfinding and Steering Using Flow Field Tiles." In: *Game AI Pro: Collected Wisdom of Game AI Professionals* (2013), p. 307.

[epoll] Linux man-pages project. *Linux Programmers Manual: epoll*. Sept. 2017.

[Fai] T. Faison. *Event-Based Programming*. apress.

[Fau03] K. Fauerby. "Improved collision detection and response." In: *DK, private communication* (2003).

[FC80] F. N. Fritsch and R. E. Carlson. "Monotone piecewise cubic interpolation." In: *SIAM Journal on Numerical Analysis* 17.2 (1980), pp. 238–246.

[Int16] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 2 (2A, 2B, 2C & 2D). 2016.

[Jon+08] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. "Cyclic dependencies in modular performance analysis." In: *Proceedings of the 8th ACM international conference on Embedded software*. ACM. 2008, pp. 179–188.

[Mic17] Microsoft Corp. *Handling and Raising Events*. 2017. URL: `https://docs.microsoft.com/en-us/dotnet/standard/events/` (visited on 09/18/2017).

[Nys14] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[OAge15] Openage Developers. *openage*. 2015. URL: `https://openage.sft.mx` (visited on 09/05/2017).

[Ont+13] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. "A survey of real-time strategy game ai research and competition in starcraft." In: *IEEE Transactions on Computational Intelligence and AI in games* 5.4 (2013), pp. 293–311.

[Pot00] D. C. Pottinger. "Terrain analysis in realtime strategy games." In: *Game Developers Conference 2000*. 2000.

[Pro16] N. T. Prof. Dr. Rüriger Westermann. *Game Physics*. Nov. 2016.

[Pyt17] Python Inc. *Asynchronous I/O, event loop, coroutines and tasks*. 2017. URL: `https://docs.python.org/3/library/asyncio.html` (visited on 08/10/2017).

[read] Linux man-pages project. *Linux Programmers Manual: read*. Sept. 2017.

[Smi13] F. Smith. *The Tech of Planetary Annihilation: ChronoCam*. 2013. URL: `https://blog.forrestthewoods.com/the-tech-of-planetary-annihilation-chronocam-292e3d6b169a` (visited on 08/10/2017).

[Stu84] D. Sturman. "Interactive Keyframe Animation of 3-D Articulated Models." In: *Proceedings of Graphics Interface '84*. GI '84. Ottawa, Ontario, Canada: National Research Council of Canada, 1984, pp. 35–40.