Game Engine Development
Pathfinding & AI

Michael Schreiber
Supervisor: Assoc.-Prof. Dipl.-Ing. Dr. Stefan Resmerita

Universität Salzburg
Fachbereich Computerwissenschaften

Abstract. Usually, every modern computer game is based on a *Game Engine*. Today, there is a great variety of *Game Engines* available, created and maintained by the so called *triple A studios*, which often can be licensed or bought by developers, to create their own games with low effort. No matter if they choose Epic´s *Unreal Engine*, Crytek´s *CryEngine* or DICE´s *Frostbite Engine*: For small game studios, this often is the only viable way to release a high quality product, within a reasonable production timeline.
In this thesis, we will inspect some of the core parts of a *Game Engine*, namely *Pathfinding* and *Artificial Intelligence (AI)*.
Pathfinding - as the name suggests - is responsible, for finding a valid path between two *Nodes*. One of the most used algorithms to achieve this, is called *A-Star*, which can be customized using different *Heuristics*. For the development of an easily expandable *AI* system, we will take a closer look at the concept of *Finite-State-Machines* and how these can be extended to *Hierarchical-Finite-State-Machines*.

Keywords: Game Engine · Pathfinding · A-Star · Artificial Intelligence · Finite-State-Machine · Hierarchical-Finite-State-Machine.

Table of Contents

1 Game Engine

1.1 Introduction [2]

The term *Game Engine* arose in the mid 1990s, when two talented developers, *John Romero* and *John Carmack*, released the game *Doom (1993)* with their company *id Software*. *John Carmack*, who was responsible for the core technology of the engine, managed to decouple major components such as audio, video (rendering system), logic and maps. As part of the whole engine, *John Romero* also developed an editor called *DoomEd* (the source code was released by him in 2015), which makes creating and editing maps more comfortable.
Soon after the release and great success of *Doom*, the benefits of decoupling all the components became visible. Other companies started to license the *Doom Engine*, to create their own games. Adapting the engine to new requirements often only took minimal changes. This was also useful for the fan community of *Doom*, as they could craft their own tools to create new maps or (game changing) mods. This was possible, due to the openness of the *Doom Engine* itself. Even though, *John Romero* and *John Carmack* left *id Software*, the company - as of today - still exists and is going to release their seventh iteration of the *id Tech* engine in 2019 (in combination with the game *Doom Eternal*) [5].

The benefits and the success of *Doom*, combined with the ability to license the engine to third party companies, redefined how games would be created by *triple A studios* in the future. Inspired by the *Doom Engine*, in 1993 *Tim Sweeney* - the founder of *Epic Games* - started to work on a project called *Unreal*, featuring the *Unreal Engine*, which soon became very popular and still is maintained and licensed by the company [6].

Besides the *Doom* and *Unreal Engine*, there is a great variety of other engines to choose from when starting to develop a game today. But in general, there is one thing to keep in mind: Most engines are bound to a specific genre. For example, the *Quake III Engine* is best used for *First Person Shooter (FPS)* games. This is due to the fact, that parts of the engine (in particular the graphical rendering) are often optimized for a certain kind of genre. This can lead to problems, where an engine, which performs superior in rendering indoor environments, could totally fail for outdoor scenarios. The main reason is, that there exist different techniques for different scenarios: Outdoor environments require an aggressive use of *Level Of Detail (LOD)* techniques, which (dynamically) reduces the details of distant objects, to optimize the performance. On the other hand, indoor environments would not require such a technique. Instead, it would be important to render all objects with maximum detail.

1.2 Architecture Layers [2]

Even though there is no archetypal *Game Engine* design, there exist common components which an engine should be providing to the user. We will take a brief look at those components, but it should be kept in mind, that the lines between the components are often blurry. Furthermore, some of these components can be developed using third party *Software-Development-Kits (SDKs)*, which are often tied to a certain *Operating System (OS)* or hardware.

OS / Drivers / Hardware
A *Game Engine* may run on multiple platforms, with different hardware (e.g. *PC*, *Xbox*, *Playstation*, ...). Typically, some kind of communication with the underlying hardware is needed, which can be accomplished using either device drivers or through *Application Programming Interfaces (APIs)* exposed by the *OS*.

Third Party SDKs
Third Party SDKs can be used to provide access to graphics hardware (e.g. *DirectX* or *OpenGL*), collision detection and physics (e.g. *Havok*, *PhysX*, *Open Dynamics Engine (ODE)*).

Platform Independence Layer
In case the *Game Engine* should run on multiple platforms, the *Platform Independence Layer* makes sure that fundamental things (e.g. *APIs*, libraries) offer consistent behavior - no matter which platform the engine is running on.

Core Support and Utility Systems
Every *Game Engine* ships with core utility classes, including (but not limited to) math libraries, parsers, debug helpers.

Resource Manager
A *Resource Manager* is responsible for accessing different types of game assets (e.g. textures, animation data, maps, ...).

Rendering
Rendering is a massive, mandatory and complex part of every *Game Engine* (especially when dealing with 3D). It is usually divided into four main categories:

- **Renderer**
  This component typically accesses the graphics device (through the *Platform Independence Layer*), to render geometric primitives, such as triangles.
- ***Scene Graph / Culling Optimizations***
  The *Scene Graph* is responsible for managing the amount of objects the *Renderer* has to process. There are different techniques to achieve this, such as *Level Of Detail (LOD)*, *Spatial Subdivision* or *Occlusion & Potentially Visible Sets (PVS)*.
- **Visual Effects**
  Depending on the rendering engine, there might be several *Visual Effects* supported, such as particle effects, dynamic shadows, decal system, post render effects (*High Dynamic Range (HDR)*, *Anti Aliasing (AA)*, ...).
- **Front End**
  The *Front End* layer is responsible for 2D graphics and includes things like *Heads Up Display (HUD)*, *User Interface (UI)*, menus, ...

Collision and Physics
Collision and Physics are usually tied together. Due to the complexity of both areas, many engines tend to use third party libraries, such as *Havok*, *PhysX* or the open source alternative *ODE*.

Animation System
There are different types of *Animation Systems* (e.g. sprite / texture animation, vertex animation, ...) but the most prevalent is skeletal animation.

Human Interface Devices
This component processes the input from the player devices - e.g. a keyboard. Besides input, there is also output that can be processed (e.g. force feedback).

Audio
The *Audio Engine* offers support for different input sound formats.

Network
Game Engines typically provide support for multiplayer. Depending on the capabilities of the engine, there might be different (supported) models, such as peer-to-peer or client-server structure.

AI
An ideal model for an abstract and extendable *AI* is a *Hierarchical-Finite-State-Machine (HFSM)* (see section 2.3), which helps to define and extend the different layers, required for creating a complex *AI*. Another part of the *AI* component is *Pathfinding*, which should be fast and reliable. There are different algorithms to achieve this, with *A-Star* being one of the most commonly used.

2 AI

2.1 Introduction

> **Definition**
>
> The term **Artificial Intelligence (AI)** is applied when a machine mimics "cognitive" functions that humans associate with other human minds, such as "learning" and "problem solving".

AI is a mandatory part in today's gaming industry and its development process is a challenging task. In *triple A studios*, specialized developers are working exclusively on *AI* systems, to increase the immersion of virtual gaming worlds. We are going to take a look at an intuitive concept, which helps to develop both, a complex but still extendable *AI* system. To achieve this, the principles of a *Finite-State-Machine (FSM)* will be explained first. Furthermore, the *FSM* will then be extended into a *HFSM*, which helps at creating an *AI* scenario (section 2.4.2).

2.2 Finite-State-Machine [1]

Definition

A **Finite-State-Machine (FSM)** is a mathematical model of computation. It can be in exactly one of a finite number of *States* and changes from one *State* to another in response to some external input. This change is called *Transition* [7].

There are many different ways to compose *FSMs*. Rather than using arithmetic syntax, we prefer to use visual syntax with circles and arrows.
Note: Our *FSM* is defined as a *Deterministic-Finte-State-Machine (DFSM)*, which means that it can only be in, and transition to, one *State* at a time.

Initial State $\bullet\!\!\rightarrow$
The *Initial State* symbol marks the starting *State* of a *FSM*. This means, upon entering a *FSM*, the *Initial State* will be the one executed first.

Transition $\rightarrow$
A *Transition* describes the change from one *State* to another, if the expression above (or beneath) the arrow evaluates to *true*. An expression consists of one or multiple *booleans*, which can be combined by using one (or more) logical operators:

- or: $\vee$
- and: $\wedge$
- negation: $\neg$

Self Transition $\circlearrowright$
A transition is called *Self Transition*, if it starts and ends at the same *State*.

Example

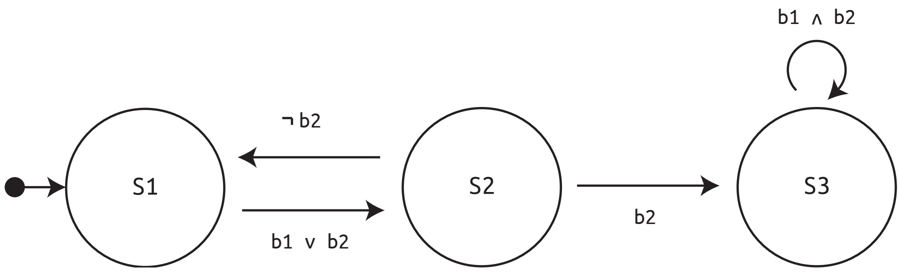

Fig. 1. Simple example of a FSM, using the notation described above

b_1 to b_n are *booleans*. S_1 to S_n are *States*, with S_1 being the initial *State*. Given the default value of all *booleans* is *false*, we now set $b_2 = true$ and start executing the *FSM* (fig. 1). The *FSM* will now start in S_1, transition from S_1 to S_2 and execute a_1, transition from S_2 to S_3 and execute a_3. In S_3 there is a possible self transition, while $b_1 = true$ and $b_2 = true$ (which will not happen, since we only set b_1 to *true* - not b_2).

2.3 Hierarchical-Finite-State-Machine

Definition

A **Hierarchical-Finite-State-Machine (HFSM)** is a *FSM*, containing one (or more) *States*, which themselves can be a *FSM*.

[9] When adding or removing *States* in a *FSM*, it is often necessary to change conditions of other *States* that have transitions to the new or old one, which can lead to (potential) errors. Instead of having every *State* transitioning to another *State*, the *HFSM* is less susceptible to those kinds of errors, by introducing the idea of encapsulation. This concept dramatically reduces the amount of transitions required, as they are now shared by multiple child *States* of a *FSM*. When creating a *HFSM* it may be helpful to start off by thinking top-down (breaking problems into smaller, modular parts) and building a tree structure.

2.4 Code [10]

Note: In the interest of simplification, we will only use the term *FSM* in this section, instead of switching between *FSM* and *HFSM*.
To demonstrate the practical use of a *FSM* in game development, we will show implementations based on the programming language *JavaScript*. Furthermore, we will use the *Lucid Engine* (*JavaScript Game Engine*, developed by the author of this thesis) and its implementation of the *FSM* (section 2.4.1), combined with a concrete scenario (section 2.4.2).

2.4.1 FSM (Lucid Engine)

2.4.1.1 Unified Modeling Language
The *Unified Modeling Language (UML)* diagram of the *Lucid Engine FSM* can be seen in figure 2.

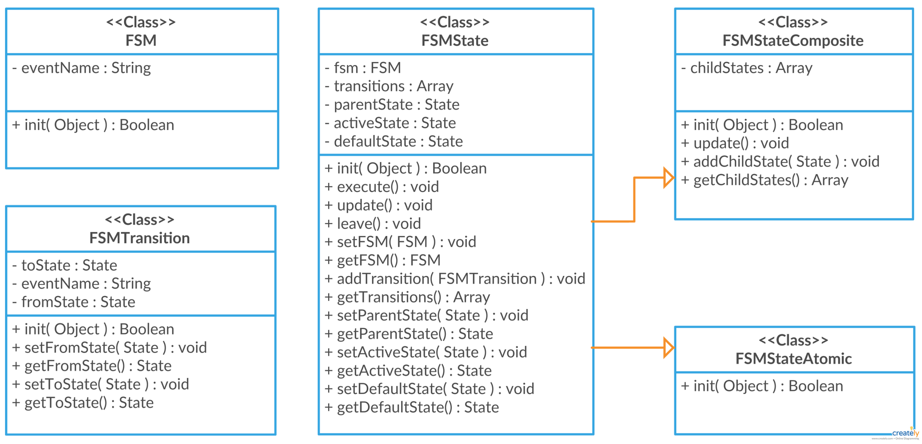

Fig. 2. UML for the code structure of the Lucid Engine FSM (graphic created with https://creately.com/)

2.4.1.2 Lucid.FSM

The *Lucid.FSM* (figure 3) should be extended, when creating a custom *FSM*. The definition of the *States* and their *Transitions* should be implemented in the *init* function (line 19). Value changes of the property *eventName* (line 11) lead to transition changes.

```
1  Lucid.FSM = Lucid.BaseComponent.extend({
     // local variables
     eventName: "", // the event name defines the current transition

     // Automatically called when instantiated.
6    init: function(config) {
       this.checkSetComponentName("Lucid.FSM");

       this._super(config);

11     return true;
     }
   });
```

Fig. 3. FSM

2.4.1.3 Lucid.FSMTransition

The *Lucid.FSMTransition* (figure 4) is used to change from one State to another, based on the defined *eventName* property and the current *Lucid.FSM* reference object's *eventName* property.

```
Lucid.FSMTransition = Lucid.BaseComponent.extend({
  // config variables and their default values
  toState: null, // [required] the state to transition to
  eventName: null, // [required] event name which triggers this transition

  // local variables
  fromState: null, // from state - this will be injected when added to a state

  // Automatically called when instantiated.
  init: function(config) {
    this.checkSetComponentName("Lucid.FSMTransition");

    this._super(config);

    if (!this.toState) {
      Lucid.Utils.error(this.componentName + "␣@␣init:␣toState␣is␣null!");
      return false;
    }

    if (!this.eventName) {
      Lucid.Utils.error(this.componentName + "␣@␣init:␣eventName␣is␣not␣defined!");
      return false;
    }

    return true;
  },

  setFromState: function(fromState) {
    this.fromState = fromState;
  },

  getFromState: function() {
    return this.fromState;
  },

  setToState: function(toState) {
    this.toState = toState;
  },

  getToState: function() {
    return this.toState;
  }
});
```

Fig. 4. FSMTransition

2.4.1.4 Lucid.FSMState

The *Lucid.FSMState* (figure 5) is the base for both: *Lucid.FSMStateComposite* and *Lucid.FSMStateAtomic*, which means they both extend *Lucid.FSMState*. The *execute* function (line 26) can be overridden to implement the custom logic (when the *State* is active), which can be seen in the *States* of the scenario (section 2.4.2).

```
Lucid.FSMState = Lucid.BaseComponent.extend({
  // config variables and their default values
  fsm: null, // [required] fsm reference object

  // local variables
  transitions: [], // array with transitions
  parentState: null, // reference to parent state
  activeState: null, // currently active state
  defaultState: null, // the default state

  // Automatically called when instantiated.
  init: function(config) {
    this.checkSetComponentName("Lucid.FSMState");

    this._super(config);

    if (!this.fsm) {
      Lucid.Utils.error(this.componentName + "␣@␣init:␣fsm␣is␣null!");
      return false;
    }

    return true;
  },

  // Override to implement logic.
  execute: function() {
  },

  // See FSMStateComposite -> update()
  update: function() {
  },

  // This method is called, if this State is left due to a Transition.
  // This also notifies recursively other (underlying) active States!
  leave: function() {
    if (this.getActiveState()) {
      this.getActiveState().leave();
    }
  },

  setFSM: function(fsm) {
    this.fsm = fsm;
  },

  getFSM: function() {
    return this.fsm;
  },

  // Adds a Transition to this and sets the
  // Transitions from State to this.
  addTransition: function(transition) {
    transition.setFromState(this);
    this.transitions.push(transition);
  },

  getTransitions: function() {
    return this.transitions;
  },

  setParentState: function(parentState) {
    this.parentState = parentState;
  },

  getParentState: function() {
    return this.parentState;
  },

  setActiveState: function(activeState) {
    this.activeState = activeState;
  },

  getActiveState: function() {
    return this.activeState;
  },

   // Sets the default State. Sets the active State to
   // default State - if not set yet.
  setDefaultState: function(defaultState) {
    this.defaultState = defaultState;

    if (!this.getActiveState()) {
      this.setActiveState(defaultState);
    }
  },

  getDefaultState: function() {
    return this.defaultState;
  }
});
```

Fig. 5. FSMState

2.4.1.5 Lucid.FSMStateComposite

The *Lucid.FSMStateComposite* (figure 6) extends *Lucid.FSMState*. It is a container *State*, which means that it does contain one or multiple *child States* and a concrete implementation of the *update* method (line 17). By default, the first *active State* is the specified *default State*.

```
Lucid.FSMStateComposite = Lucid.FSMState.extend({
  // local variables
  childStates: [], // array with child states

  // Automatically called when instantiated.
  init: function(config) {
    this.checkSetComponentName("Lucid.FSMStateComposite");

    this._super(config);

    return true;
  },

  // Checks the currently active State Transitions and (if necessary)
  // changes to a new active State. This happens, by comparing the Transitions
  // eventName property with the FSM reference Object eventName property.
  update: function() {
    if (!this.getActiveState()) {
      return false;
    }

    // CASE 1: transition required for this active state

    // fetch transitions from this active state
    transitions = this.getActiveState().getTransitions();

    // sanity check
    if (transitions.length > 0) {

      for (i = 0; i < transitions.length; ++i) {
        var transition = transitions[i];

        if (transition && this.getFSM().eventName == transition.eventName) {
          // get fromState and new toState from transition
          fromState = transition.getFromState();
          toState = transition.getToState();

          // leave fromState (possible recursion)
          fromState.leave();

          // set toState as currently active state
          this.setActiveState(toState);

          // set toState active state to its default state (if available)
          if (toState.getDefaultState()) {
            toState.setActiveState(toState.getDefaultState());
          }

          // execute / update the new toState (possible recursion)
          toState.execute();
          toState.update();

          // leave here, so we dont end up in CASE 2
          return;
        }
      }
    }

    // CASE 2: there was no transition - just execute this active state.
    // NOTE: if this active state changed the eventName, we dont want to update
    // it (the active state), as its childs could possibly change the eventName again!

    var tmpEventName = this.getFSM().eventName;
    this.getActiveState().execute();
    if (this.getFSM().eventName == tmpEventName) {
      this.getActiveState().update();
    }
  },

  // Adds a child State to this and sets the
  // States parent State to this.
  addChildState: function(state) {
    state.setParentState(this);
    this.childStates.push(state);
  },

  getChildStates: function() {
    return this.childStates;
  }
});
```

Fig. 6. FSMStateComposite

2.4.1.6 Lucid.FSMStateAtomic

The *Lucid.FSMStateAtomic* (figure 7) extends *Lucid.FSMState*. It is a final *State*, which means it does not contain any child *States*.

```
Lucid.FSMStateAtomic = Lucid.FSMState.extend({

  // Automatically called when instantiated.
  init: function(config) {
    this.checkSetComponentName("Lucid.FSMStateAtomic");

    this._super(config);

    return true;
  }
});
```

Fig. 7. FSMStateAtomic

2.4.2 Scenario (Game)

We define a scenario, where two hostile *Entities* want to attack each other. Each *Entity* has the following *Atomic States*:

- **Idle**
  The *Entity* is idling, waiting for a hostile *Entity* to get into the line of sight, i.e. that there are no obstacles in between the two *Entities*.
- **Approach**
  The *Entity* is trying to approach a hostile *Entity*, as long as the other *Entity* is in line of sight.
- **Attack**
  The *Entity* is attacking a hostile *Entity*, as long as the other *Entity* is in range.
- **Heal**
  The *Entity* is recovering health. This should happen, as soon as the health falls below 30%.

The idea is, that an *Entity Idles*, as long as it does not have line of sight to a hostile *Entity*. As soon as line of sight is available, it should *Approach* the hostile *Entity* until it reaches the minimum distance for an *Attack*.
But the most important task is to survive, which means whenever the health drops below 30% the *Entity Heals* itself.
The full schema can be seen in figure 8.

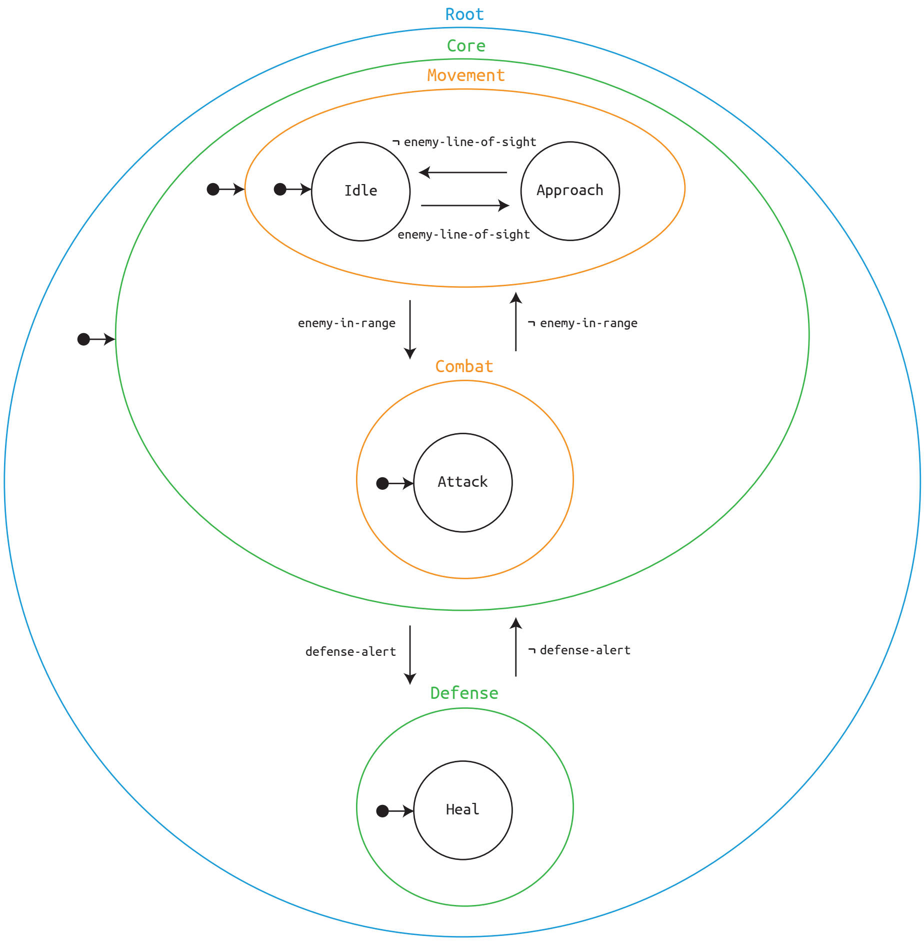

Fig. 8. FSM for the scenario given

2.4.2.1 Game.FSM

The *Game.FSM* (figure 9) extends *Lucid.FSM*. It is used in a *Lucid.AI* object, which can be attached to *Entities* and ultimately simulates artificial intelligence, based on the concept of the *Game.FSM*. All *States* (line 21-64) and their *Transitions* (line 69-76) - as defined in figure 8 - are implemented in the *init* function. The *renderUpdate* function (line 87) is called continuously by the *Lucid Engine*, updating (recursively) the *States* and *Transitions* of the *Game.FSM* - if required. The *Lucid.AI* object reference (line 3) is required and acts as a util object (for *AI* specific calculations).

Note: *Game.FSM.EVENTS* is an object with string constants (for the *eventName*).

```
Game.FSM = Lucid.FSM.extend({
  // config variables and their default values
  ai: null, // [required] reference to the ai

  // local variables
  root: null, // root state

  init: function(config) {
    this.componentName = "Game.FSM";

    this._super(config);

    if (!this.ai) {
      Lucid.Utils.error(this.componentName + " @ init: ai is null!");
      return false;
    }

    /////////////////////////
    // step 1 -> setup states:

      // root
      this.root = new Lucid.FSMStateComposite({ componentName: this.componentName + ".Root", fsm: this });

      // composite states (of root):
        // a) core
        var core = new Game.FSM.Root.Core({ componentName: "Core", fsm: this });

        // composite states (of core):
          // a.a) movement
          var movement = new Game.FSM.Root.Core.Movement({ componentName: "Movement", fsm: this });

          // atomic states (of movement)
            var idle = new Game.FSM.Root.Core.Movement.Idle({ componentName: "Idle", fsm: this });
            var approach = new Game.FSM.Root.Core.Movement.Approach({ componentName: "Approach", fsm: this });

          movement.addChildState(idle);
          movement.addChildState(approach);
          movement.setDefaultState(idle);

          // a.b) combat
          var combat = new Game.FSM.Root.Core.Combat({ componentName: "Combat", fsm: this });

          // atomic states (of combat)
            var attack = new Game.FSM.Root.Core.Combat.Attack({ componentName: "Attack", fsm: this });

          combat.addChildState(attack);
          combat.setDefaultState(attack);

        core.addChildState(movement);
        core.addChildState(combat);
        core.setDefaultState(movement);

        // b) defense
        var defense = new Game.FSM.Root.Defense({ componentName: "Defense", fsm: this });

        // atomic states (of defense)
          var heal = new Game.FSM.Root.Defense.Heal({ componentName: "Heal", fsm: this });

        defense.addChildState(heal);
        defense.setDefaultState(heal);

      this.root.addChildState(core);
      this.root.addChildState(defense);
      this.root.setDefaultState(core);

    /////////////////////////
    // step 2 -> setup transitions:

    idle.addTransition(new Lucid.FSMTransition({ toState: approach, eventName: Game.FSM.EVENTS.ENEMY_LINE_OF_SIGHT }));
    approach.addTransition(new Lucid.FSMTransition({ toState: idle, eventName: Game.FSM.EVENTS.NOT_ENEMY_LINE_OF_SIGHT }));

    movement.addTransition(new Lucid.FSMTransition({ toState: combat, eventName: Game.FSM.EVENTS.ENEMY_IN_RANGE }));
    combat.addTransition(new Lucid.FSMTransition({ toState: movement, eventName: Game.FSM.EVENTS.NOT_ENEMY_IN_RANGE }));

    core.addTransition(new Lucid.FSMTransition({ toState: defense, eventName: Game.FSM.EVENTS.DEFENSE_ALERT }));
    defense.addTransition(new Lucid.FSMTransition({ toState: core, eventName: Game.FSM.EVENTS.NOT_DEFENSE_ALERT }));

    /////////////////////////
    // step 3 -> start:

    this.root.setDefaultState(core);

    return true;
  },

  // simulation update
  renderUpdate: function(delta) {
    this.root.update();
  },

  setAI: function(ai) {
    this.ai = ai;
  },

  getAI: function() {
    return this.ai;
  }
});
```

Fig. 9. Game.FSM

2.4.2.2 Game.FSM.Root.Core

The *Game.FSM.Root.Core* (figure 10) extends *Lucid.FSMStateComposite*. It specifies, that in case when the *originEntity's* (the *Entity*, which the *Lucid.AI* and *Game.FSM* is attached to) *healthPoints* drop below 30%, a change of the *eventName* to *DEFENSE_ALERT* happens (which results in a *Transition*).

```
Game.FSM.Root.Core = Lucid.FSMStateComposite.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();

    // if lower than defined percent ...
    if (originEntity.healthPointsCurrent / originEntity.healthPointsMaximum < 0.3) {
      // ... change eventName to DEFENSE_ALERT
      this.fsm.eventName = Game.FSM.EVENTS.DEFENSE_ALERT;
    }
  }
});
```

Fig. 10. Game.FSM.Root.Core

2.4.2.3 Game.FSM.Root.Core.Movement

The *Game.FSM.Root.Core.Movement* (figure 11) extends *Lucid.FSMStateComposite*. It checks for hostile *Entities* within the *originEntitity's* line of sight and changes the *eventName* to *ENEMY_IN_RANGE*.

```
Game.FSM.Root.Core.Movement = Lucid.FSMStateComposite.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();
    // // try to get the first (closest) hostile entity in line-of-sight ...
    var closestHostileEntityInLineOfSight = this.fsm.ai.getHostileEntitiesInLineOfSight()[0];

    // ... if available ...
    if (closestHostileEntityInLineOfSight) {
      // ... check if its within the originEntity minimumAttackRange ...
      if (Lucid.Math.getDistanceBetweenTwoEntities(closestHostileEntityInLineOfSight, originEntity)
        <= originEntity.minimumAttackRange) {
        // ... change eventName to ENEMY_IN_RANGE
        this.fsm.eventName = Game.FSM.EVENTS.ENEMY_IN_RANGE;
      }
    }
  }
});
```

Fig. 11. Game.FSM.Root.Core.Movement

2.4.2.4 Game.FSM.Root.Core.Movement.Idle

The *Game.FSM.Root.Core.Movement.Idle* (figure 12) extends *Lucid.FSMStateAtomic*. It changes the *eventName* to *ENEMY_LINE_OF_SIGHT*, if there exists a hostile *Entity* within the *originEntitity's* line of sight.

```
Game.FSM.Root.Core.Movement.Idle = Lucid.FSMStateAtomic.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();
    // try to get the first (closest) hostile entity in line-of-sight ...
    var closestHostileEntityInLineOfSight = this.fsm.ai.getHostileEntitiesInLineOfSight()[0];

    // ... if available ...
    if (closestHostileEntityInLineOfSight) {
      // ... change eventName to ENEMY_LINE_OF_SIGHT
      this.fsm.eventName = Game.FSM.EVENTS.ENEMY_LINE_OF_SIGHT;
    }
  }
});
```

Fig. 12. Game.FSM.Root.Core.Movement.Idle

2.4.2.5 Game.FSM.Root.Core.Movement.Approach

The *Game.FSM.Root.Core.Movement.Approach* (figure 13) extends *Lucid.FSMStateAtomic*. It checks for a hostile *Entity* in line of sight and approaches it (by setting the *originEntity's* path). If no valid hostile *Entity* was found, it changes the *eventName* to *NOT_ENEMY_LINE_OF_SIGHT*. Additionally, it overrides the *leave* function (line 45), by implementing a reset of the *originEntity's* path.

```
1  Game.FSM.Root.Core.Movement.Approach = Lucid.FSMStateAtomic.extend({
2    init: function(config) {
       this._super(config);

       // check / set map reference
       this.checkSetMap();
7
       return true;
     },

     execute: function() {
12     var originEntity = this.fsm.ai.getOriginEntity();
       // try to get the first (closest) hostile entity in line-of-sight ...
       var closestHostileEntityInLineOfSight = this.fsm.ai.getHostileEntitiesInLineOfSight()[0];

       // ... if available ...
17     if (closestHostileEntityInLineOfSight) {
         var originEntityGridIndices = Lucid.Math.getEntityToGridIndices(
           originEntity, this.map.tileSize);
         var targetEntityGridIndices = Lucid.Math.getEntityToGridIndices(
           closestHostileEntityInLineOfSight, this.map.tileSize);
22
         // ... approach it
         Lucid.Pathfinding.findPath(
           originEntityGridIndices[0],
           originEntityGridIndices[1],
27         targetEntityGridIndices[0],
           targetEntityGridIndices[1],
           function(path) {
             if (path) {
               this.fsm.ai.getOriginEntity().setPath(path);
32           }
           }.bind(this)
         );

         Lucid.Pathfinding.calculate();
37     }
       // ... if not available ...
       else {
         // ... change eventName to NOT_ENEMY_LINE_OF_SIGHT
         this.fsm.eventName = Game.FSM.EVENTS.NOT_ENEMY_LINE_OF_SIGHT;
42     }
     },

     leave: function() {
       this._super();
47
       // stop movement
       this.fsm.ai.getOriginEntity().setPath(null);
     }
   });
```

Fig. 13. Game.FSM.Root.Core.Movement.Approach

2.4.2.6 Game.FSM.Root.Core.Combat

The *Game.FSM.Root.Core.Combat* (figure 14) extends *Lucid.FSMStateComposite*. It changes the *eventName* to *NOT_ENEMY_IN_RANGE*, in case there is no hostile *Entity* within the *originEntitity's minimumAttackRange*.

```
Game.FSM.Root.Core.Combat = Lucid.FSMStateComposite.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();

    // try to get the first (closest) hostile entity in line-of-sight ...
    var closestHostileEntityInLineOfSight = this.fsm.ai.getHostileEntitiesInLineOfSight()[0];

    // ... if available ...
    if (closestHostileEntityInLineOfSight) {
      // ... check if its outside of the originEntity minimumAttackRange ...
      if (Lucid.Math.getDistanceBetweenTwoEntities(closestHostileEntityInLineOfSight, originEntity)
        > originEntity.minimumAttackRange) {
        // ... change eventName to NOT_ENEMY_IN_RANGE
        this.fsm.eventName = Game.FSM.EVENTS.NOT_ENEMY_IN_RANGE;
      }
    }
  }
});
```

Fig. 14. Game.FSM.Root.Core.Combat

2.4.2.7 Game.FSM.Root.Core.Combat.Attack

The *Game.FSM.Root.Core.Combat.Attack* (figure 15) extends *Lucid.FSMStateAtomic*. It applies damage to the first hostile *Entity*, within the *originEntity's minimumAttackRange*.

```
Game.FSM.Root.Core.Combat.Attack = Lucid.FSMStateAtomic.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();

    // try to get the first (closest) hostile entity in line-of-sight ...
    var closestHostileEntityInLineOfSight = this.fsm.ai.getHostileEntitiesInLineOfSight()[0];

    // ... if available ...
    if (closestHostileEntityInLineOfSight) {
      // ... check if its within of the originEntity minimumAttackRange
      if (Lucid.Math.getDistanceBetweenTwoEntities(closestHostileEntityInLineOfSight, originEntity)
        <= originEntity.minimumAttackRange) {
        closestHostileEntityInLineOfSight.healthPointsCurrent = Math.max(
          0, closestHostileEntityInLineOfSight.healthPointsCurrent - 0.5);
      }
    }
  }
});
```

Fig. 15. Game.FSM.Root.Core.Combat.Attack

2.4.2.8 Game.FSM.Root.Core.Defense

The *Game.FSM.Root.Core.Defense* (figure 16) extends *Lucid.FSMStateComposite*. It checks if the *originEntity* has *healthPoints* greater than or equal to 30%. If the condition is met, it changes the *eventName* to *NOT_DEFENSE_ALERT*.

```
Game.FSM.Root.Defense = Lucid.FSMStateComposite.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();

    // higher (or equal) than defined percent health?
    if (originEntity.healthPointsCurrent / originEntity.healthPointsMaximum >= 0.3) {
      this.fsm.eventName = Game.FSM.EVENTS.NOT_DEFENSE_ALERT;
    }
  }
});
```

Fig. 16. Game.FSM.Root.Core.Defense

2.4.2.9 Game.FSM.Root.Core.Defense.Heal

The *Game.FSM.Root.Core.Defense.Heal* (figure 17) extends *Lucid.FSMStateAtomic*. It implements the recovery of missing *healthPoints*.

```
Game.FSM.Root.Defense.Heal = Lucid.FSMStateAtomic.extend({
  execute: function() {
    var originEntity = this.fsm.ai.getOriginEntity();

    // recover healthPoints
    originEntity.healthPointsCurrent += 0.1;
  }
});
```

Fig. 17. Game.FSM.Root.Core.Defense.Heal

3 Pathfinding

Definition

The term **Pathfinding** means, finding a valid path between two **Nodes** in a **Graph**.

3.1 Graph & Nodes

A *Graph* is an array of *Nodes*, where each *Node* usually contains the properties seen in figure 18.

```
OBJECT Node WITH x, y, g, f, h, closed, visited, parent {
  x        // x position in the grid
  y        // y position in the grid
  g        // stores the costs so far
  f        // priority value
  h        // A-Star only: heuristic costs
  closed     // indicates if Node is closed
  visited    // indicates if Node was already visited
  parent     // reference to parent Node
}
```

Fig. 18. Properties of a Node

3.2 Map to Graph [4]

There exist different concepts of how to represent the *Map* as a *Graph* for the process of *Pathfinding*. In this thesis two methods will be shown, which can be seen in figures 19 and 20. Depending on the used concept, it is important to choose the proper *Heuristic* (see section 3.3.2).

On the one hand, representing the *Map* as a *Grid* is preferred in *2D Top-Down Games*, where the *Entities* move from the center of one *Tile* to another. In this case there could be non-walkable *Tiles*, but also *Tiles* with different weights (e.g. water, which slows down the *Entity*).

On the other hand, using *Waypoints* is best used for *3D Games*. The *Waypoints* usually must be placed by hand on the walkable surface of the *Map*. For a *Pathfinding* algorithm like *A-Star* this can be beneficial, as the number of *Nodes* determines the speed of the search.

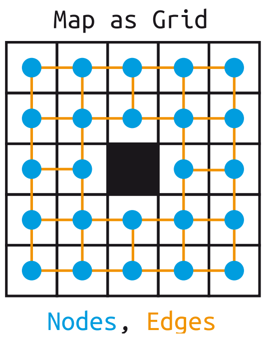

Fig. 19. Map as a Grid, using Tiles as Nodes

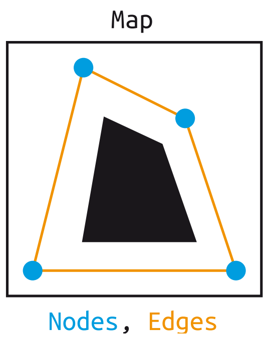

Fig. 20. Map with Waypoints as Nodes

3.3 A-Star

3.3.1 Algorithm

Figure 21 shows an example implementation of an *A-Star Pseudo Code*. An important formula of *A-Star* is $f = g + h$, where f is the total cost of the *Node*, g is the distance cost between the *current* and the *start Node* and h is the distance cost from the *current* to the *end Node* (determined by the *Heuristic* in section 3.3.2). If h is not taken into account (or simply set to zero), the algorithm would basically end up being a *Dijkstra* algorithm. The following steps represent the most important tasks in the algorithm and are corresponding to figure 21:

1. Add the *startNode* to the *nodesToVisit* (line 6).
2. Repeat:
   (a) Set *current* to the *Node* with the lowest f cost (line 9) and set its *closed* property to *true* (line 10).
   (b) For each *neighbor* of *current* (line 17), do one (or more) of the following step(s):
      i. If the property *closed* is set to *true* (line 21), continue with the next *neighbor*.
      ii. Calculate the *newG* cost (line 22), based on the position of the *neighbor* and the *current Node*. This means, *newG* is the cost, which is required for moving from *neighbor* to *current*.
      iii. (line 24) If the *neighbor* has not been *visited* yet or if the *newG* costs are smaller than the *current* g costs (which means that this is a better path), set / recalculate the new g distance and the f cost (line 29-31) and set the *parent* (line 33).

Note: Depending on the property *visited*, it may be required to update the sorting in the *Priority Queue* (line 39).

(c) End condition(1/2):

i. If the next *Node* to visit equals the *end Node* (line 13) we found a valid path. In this case we need to return a path array in reversed order. By using the property *parent* of the *current Node* we can easily back trace (and store) the reversed path, until we get to the *start Node*.

3. End condition (2/2), no valid path could be found. In this case, return *NIL* (line 44).

```
1  PROCEDURE AStar WITH start, end RETURNS Array with reversed path Nodes OR NIL {
2    // heap of nodes sorted by the "f" property value (ascending)
3    nodesToVisit is a Priority Queue
4    start.g = 0
5    start.f = 0
6    add start to nodesToVisit
7
8    WHILE nodesToVisit IS NOT empty {
9      get current from nodesToVisit
10     current.closed = true
11
12     // end condition
13     IF current x, y IS SAME AS end x, y
14       RETURN Array with reversed path Nodes
15
16     // inspect all neighbors
17     FOR EACH neighbor IN collected neighbor Nodes of current {
18       IF neighbor.closed
19         CONTINUE
20
21       // costs may differ, based on straight or diagonal movement
22       newG = costs for move
23
24       IF neighbor.visited IS false OR newG < neighbor.g {
25         // set heuristic
26         IF neighbor.h IS NIL
27           set neighbor.h to heuristic costs with neighbor x, y and end x, y
28
29         neighbor.g = newG
30         // f = g + h
31         neighbor.f = neighbor.g + neighbor.h
32         // set parent reference
33         neighbor.parent = current
34
35         IF neighbor.visited IS false
36           neighbor.visited = true
37           add neighbor to nodesToVisit
38         ELSE
39           update nodesToVisit sorting
40       }
41     }
42   }
43
44   RETURN NIL
45 }
```

Fig. 21. A-Star algorithm [8]

3.3.2 Heuristic [3]

The worst case runtime of *A-Star* is $O(n^2)$, with $n \in Nodes$. In order to get a better runtime, one should use *Heuristics*. There are many different *Heuristics* for different types of *Pathfinding* and their main purpose is to give an estimate of the length of the shortest path. One can think of it like a compass, pointing towards the target destination.
Note: Referring to the pseudo code of the *Heuristics*, *STRAIGHT_COST* usually has a value of 1 and *DIAGONAL_COST* has a value of $\sqrt{2}$.

Manhattan

Manhattan is the standard *Heuristic* when representing the *Map* as a *Grid* (figure 19), with an *Entity* that can only move in four directions (horizontal, vertical).

```
PROCEDURE Manhattan WITH nodeX, nodeY, goalX, goalY RETURNS cost {
  dx = | nodeX - goalX |
  dy = | nodeY - goalY |

  // compute number of steps you take
  REUTRN STRAIGHT_COST * ( dx + dy )
}
```

Fig. 22. Heuristic Manhattan - straight movement

Octile

Octile is best used when representing the *Map* as a *Grid* (figure 19), with an *Entity* that can move in eight directions (horizontal, vertical, diagonal).

```
PROCEDURE Octile WITH nodeX, nodeY, goalX, goalY RETURNS cost {
  dx = | nodeX - goalX |
  dy = | nodeY - goalY |

  // compute number of steps you take if you cant take a diagonal
  result = STRAIGHT_COST * ( dx + dy )

  // there are minimum( dx, dy ) diagonal steps and each one costs DIAGONAL_COST
  // but saves you 2 * STRAIGHT_COST
  RETURN result + ( DIAGONAL_COST - 2 * STRAIGHT_COST ) * minimum( dx, dy )
}
```

Fig. 23. Heuristic Octile - straight and diagonal movement

Euclidean

Euclidean is best used when the *Entity* can move at any angle (figure 20), e.g. in a non *Grid* environment.

```
PROCEDURE Euclidean WITH nodeX, nodeY, goalX, goalY RETURNS cost {
  dx = | nodeX - goalX |
  dy = | nodeY - goalY |

  // straight line distance
  RETURN STRAIGHT_COST * sqrt( dx * dx + dy * dy )
}
```

Fig. 24. Heuristic Euclidean - straight line distance

3.4 Comparison

This test is a comparison between a classical *Dijkstra* algorithm (figure 25) and the *A-Star* algorithm, using two different *Heuristics*: *Manhattan* (figure 26) and *Octile* (figure 27). The *Node* in the center (red) is the start *Node* and the one in the upper right corner is the end *Node* (green). The text on top of each *Node* indicates the step index.

Neighbor inspection order: top → left → right → bottom → (if eight directions) → top-left → top-right → bottom-left → bottom-right

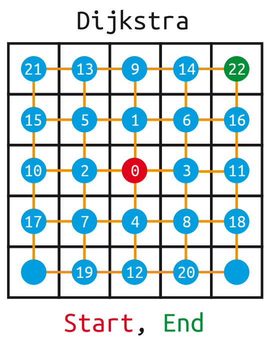

Fig. 25. Dijkstra requires 22 steps getting from start to end *Node*

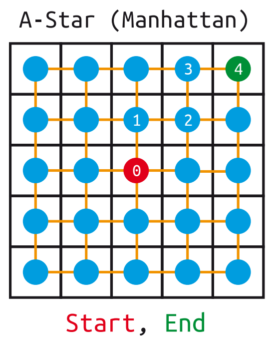

Fig. 26. A-Star (Manhattan) requires 4 steps getting from start to end *Node*

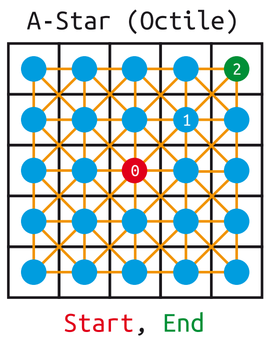

Fig. 27. A-Star (Octile) requires 2 steps getting from start to end *Node*

3.5 Lucid Engine [10]

Using the *Lucid Engine*, we can easily display the *Pathfinding* (live) results of the *A-Star* algorithm, by setting *Lucid.Debug.setPathfinding(true)* (figure 28).

Setup:

- using *Octile Heuristic*
- red *Node* is the start *Node*
- green *Node* is the end *Node*
- corner cutting (of obstacles) is disabled
- bushes are obstacles
- g, f, h as explained in section 3.3.1
- # is the index of the result path

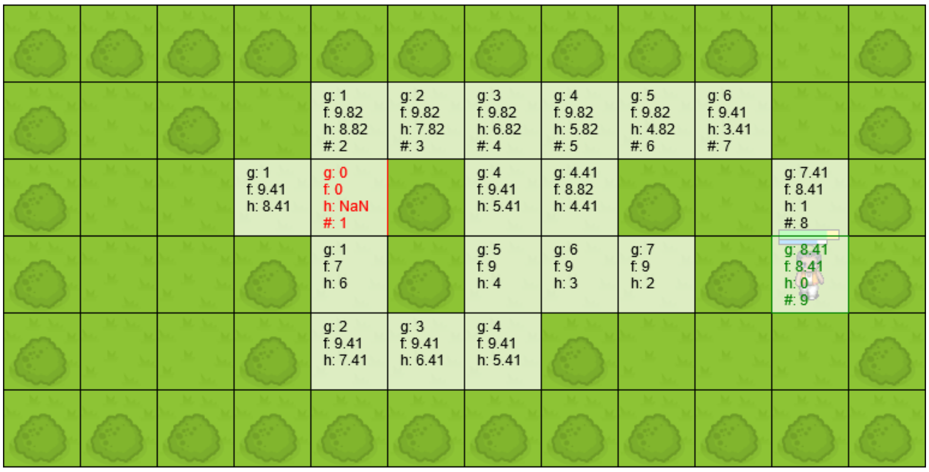

Fig. 28. Lucid Engine A-Star (Octile) Pathfinding (debug)

4 Conclusion

This thesis should give the reader a good understanding of the complex process of creating a *Game Engine*. In the early stages of game development, programmers often had to spend most of their time, working on time-consuming components (e.g. the *Renderer*). Today, developers can pick whatever *Game Engine* best fits their needs and simply adjust parameters or use additional scripts to extend it, which often does not even require a high level of programming knowledge. In the mean time, the designers can already start creating the 2D / 3D models and put them together with the help of the level editor. Due to the modular nature of such engines, updates for it can be applied, even during the game design process. But besides using such engines and especially as a computer scientist, the task of developing the engine itself (or parts of it) is an interesting topic. Even though this thesis only scratches the surface, it helps to start off with *AI* development, using *HFSMs*. Furthermore, when taking a look at the results of the *Pathfinding* comparison (see figures 25, 26, 27), it is easy to see why *A-Star* is one of the standard algorithms to use.

5 Acknowledgements

I want to thank all my colleagues, friends and especially Assoc.-Prof. Dipl.-Ing. Dr. Stefan Resmerita, for giving me important feedback and guidance during the process of creating this thesis. I thoroughly enjoyed working on the thesis and the *Lucid Engine*.

References

1. E. A. Lee and S. A. Seshia, Introduction to Embedded Systems - A Cyber-Physical Systems Approach, Second Edition, MIT Press, 2017.
2. J. Gregory, Game Engine Architecture, Second Edition, Taylor and Francis Group (LLC), 2009.
3. Amit Patel, Heuristics, http://theory.stanford.edu/%7Eamitp/GameProgramming/Heuristics.html
4. Amit Patel, Grid pathfinding optimizations, https://www.redblobgames.com/pathfinding/grids/algorithms.html
5. Wikipedia, id Tech 7, https://en.wikipedia.org/wiki/Id%5FTech%5F7
6. Wikipedia, Unreal Engine, https://en.wikipedia.org/wiki/Unreal%5FEngine
7. Wikipedia, Finite-state machine, https://en.wikipedia.org/wiki/Finite%2Dstate%5Fmachine
8. Xueqiao (Joe) Xu, Github, https://github.com/qiao/PathFinding.js/blob/master/src/finders/AStarFinder.js
9. Kyong-Sok (KC) Chang, David Zhu , 2.4 Hierarchical Finite State Machine (HFSM) & Behavior Tree (BT), https://web.stanford.edu/class/cs123/lectures/CS123%5Flec08%5FHFSM%5FBT.pdf
10. Michael Schreiber, Github, https://github.com/OxiAction/Lucid-Engine/