IT-UNIVERSITETET I KØBENHAVN

Frontpage for written work/project

ITU student	**EBUSS student**
☑ **Course**	☐ **Course**
☐ **Thesis**	☐ **Thesis**
☐ **Project** (with project agreement)	☐ **Project**
	☐ Summerproject
	☐ 4-weeks project
	☐ 12-weeks project
	☐ 16-weeks project

Title of course, project or thesis:

Game Engines

Course manager/Supervisor(s):

Mark Jason Nelson

Name(s):	Birthday and year:	ITU-mail:
1. Emil Erik Hansen	14-06-85	emha@
2. Julian Møller	18-03-87	jumo@
3. Mads Johansen	22-05-85	madj@
4. René Bech Korsgaard	28-05-84	reko@
5. Steen Nordsmark Pedersen	15-02-87	snop@

Courses with e-portfolio:

Link to e-portfolio: ______

Game Engines

Documentation of Engine

Written by:

Emil Erik Hansen	14-06-85	emha@itu.dk
Julian Møller	18-03-87	jumo@itu.dk
Mads Johansen	22-05-85	madj@itu.dk
René Bech Korsgaard	28-05-84	reko@itu.dk
Steen Nordsmark Pedersen	15-02-87	snop@itu.dk

Group: **Warm Tenderized Fudge**

IT University of Copenhagen
December 2011

Contents

List of Figures

Formalities 1

The engine was the product of the course *Game Engines*, at the *IT University of Copenhagen*, during the fall semester of 2011.

The engine and documentation is developed and written according the requirements specified on the course website[1].

1.1 Engine Requirements

To use the engine, the following is required:

- **SDL** (1.2.14) in `C:\SDL\SDL-1.2.14`
- **GLEW** (1.7.0) in `C:\GLEW\glew-1.7.0`
- **Assimp** (2.0.863) in `C:\ASSIMP\assimp--2.0.863-sdk`

It is also very important to notice that since the engine writes files upon shutting down. As such, errors are to be expected if the engine is run from a CD.

1.2 Engine Controls

In the current form, the following can be used to interact with the engine. Examples of some of these can be seen in section 4.3, page 18.

- **Keyboard, W**: Locks or unlocks the mouse to the window.
- **Keyboard, ESC**: Exits the engine.
- **Mouse, Movement**: Pans the camera in the given direction.
- **Mouse, Clicking 1**: Moves the player to the given position.
- **Mouse, Clicking 2**: Clicking the white *HUD* spawns a box.

[1] `https://blog.itu.dk/MGAE-E2011/project-requirements/`.

Engine Description

2

This section will describe the main scope behind the engine, as well as the design rationales behind the choices that were made.

2.1 Engine Scope

The engine targets the "Action `RTS`[1]"-genre, more commonly known as "`MOBA`"[2]. This includes games like *Heroes of Newerth* and *League of Legends*, both heavy inspired by the *WarCraft III* modification *Defense of the Ancients*[3].

To make our engine achieve the requirements of the genre, it was important to have a clear idea of what the expected result should look like. As a good reference, see the example screenshot in figure 2.1, page 3.

As can be seen, the basic graphics are fairly simple. There has to be a flat ground-level upon which the player can be controlled around, along with objects he cannot pass. The avatar of the player has to be able to get from `A` to `B`, and be able to pathfind around various obstacles. On top of that, there has to be an interactable heads-up display. The camera will do limited movement, staying with a fixed angle.

This puts the main focus of the engine on effective handling of mouse events, collision-detection and pathfinding. An example of how it looks like can be seen in figure 2.2, page 4. Further examples can be seen in section 4.3, page 18.

Engine Structure

We ended out structuring our engine like shown in figure 2.3, page 5.

The engine contains each of the blue subsystems (window, settings, events, performance monitor, memory, graphics, physics).

[1]Real-Time Strategy.

[2]Multiplayer Online Battle Arena. `http://en.wikipedia.org/wiki/Dota_(genre)`

[3]`http://en.wikipedia.org/wiki/Defense_of_the_Ancients`

Figure 2.1: Engine Scope: Inspirational screenshot from *Defense of the Ancients*.

The graphics subsystem contains handling of the HUD, 3D models, materials and a media manager, that handles loading of images and model files.

Some components (the path planner, scene description graph and game objects) are heavily reliant on both the physics and graphics subsystems, and are therefore placed on the intersection between the two systems in the diagram.

2.2 Design Rationales

This section covers what decisions was put behind the major parts of the engine: The *graphics-* and *physics systems* and the *path planner*.

Graphics

Displaying graphics on the screen is the most important part of every game engine, and as such some design decisions would have to be made as to how to both import assets as well as how to render them.

As far as importing assets both models and textures would have to be imported, and two options were considered: using a library and wrap it into the

Figure 2.2: Engine Scope: Basic view of our engine.

engine or write custom importers. They both have their distinct advantages. Using a library will save precious time on having to implement an importer as well as allowing us to import a wide range of file formats. On the other hand creating a custom loader would let us control exactly how the loading occurs and optimize it for performance. Since it was deemed interesting and educational to create a custom loader and since it would allow for better performance, this approach was chosen, and a custom `.TGA`-texture loader was created and is part of the engine. We started on a custom `MS3D`-loader, but ended up utilizing a library for all of the model loading. The library of choice was *Assimp*, since it was fairly simple to use and supported a lot of different file formats.

When it comes to method of rendering both dynamic lists and vertex buffers were considered. As the only advantage of using dynamic list were deemed to be ease of use, vertex buffers were chosen for their supposedly better performance. It was quite a struggle to get them well integrated into the project. It was first intended to create one vertex buffer per shader, but since we initially had problems implementing the shaders, we decided to drop them for lack of time. Thus, we ended up with one big vertex buffer, since it

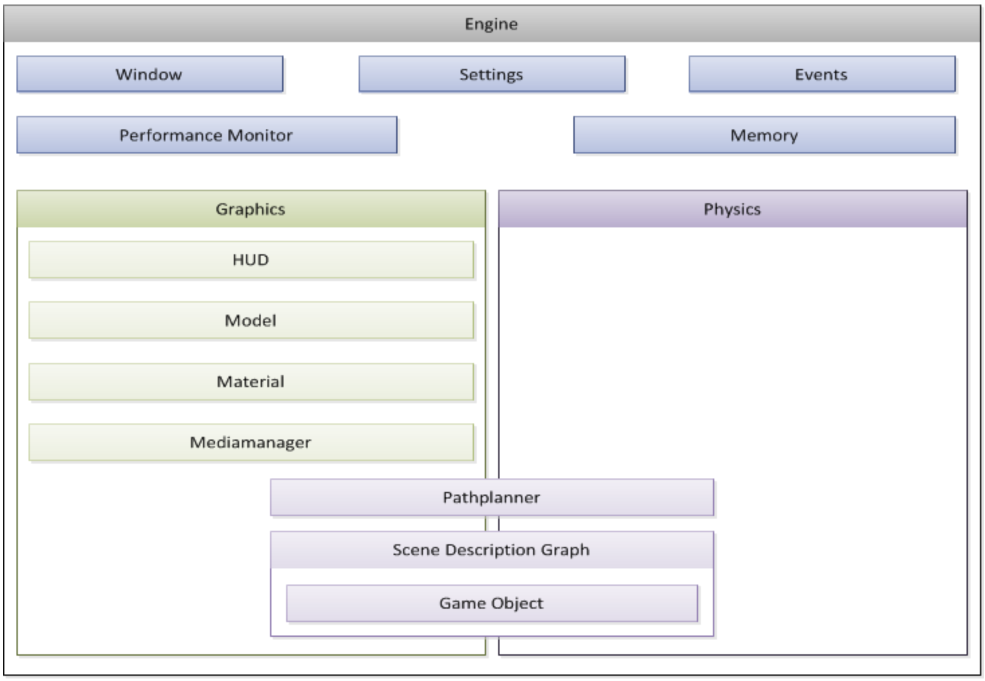

Figure 2.3: Engine Structure: Diagram showing the structure.

would theoretically yield the best performance. However, it was soon realized that it was much simpler to work with one vertex buffer per object, and as such this solution was implemented instead, even though the performance might be a bit worse.

Physics System

The physics system in an *RTS* is an ideal place to cut corners. Since the genre originally emerged as a *2D* game, all game play in *RTS* traditionally are in *2D* even though the graphics are *3D*. Flying units can even be implemented using a *2D* physics system by just adding levels to the physics objects and then checking if objects are on the same level before triggering a collision. Thus, for simplicity it was chosen to implement a *2D* physics system. Furthermore it was chosen to let the physics system control all movement.

Path Planner

Pathfinding is an essential component of any *RTS* game, and we also needed it in our engine. This could be done in a number of different ways (navigation mesh, grid based path finding, using an `A*` algorithm, breadth first search, etc.). Since the world of *RTS* games are typically fairly open we decided to make the path finding grid based, because of the simplicity. The precision of the pathing can then be altered via the number of grid divisions. However, the precision is reversely proportional with the performance of the path finding. One might argue that for performance it would be better to implement a navigation mesh. However, the simplicity of setting up a grid based path finding is much simpler for the user of the engine than setting up an entire navigation mesh. The user just has to set objects into the scene and send them to the path planner, that then takes them into consideration. After these simple steps, the pathfinding is functional. With a navigation mesh the user would have to define each node by defining a convex polygon. This is arguable a lot more cumbersome.

`A*` was chosen as the actual path finding algorithm, since it allows for implementing different terrain that is harder traversable, and still finding the shortest path in terms of time spent getting there.

2.3 Features

To make our game function optimally for the specific game genre, the main features are:

- An effective event system, both sendable by input devices and game objects.
- A `HUD` that the mouse can interact with.
- Pathfinding.

Implementation 3

This section gives a more detailed description of the various subparts of the engine, based on the list given in Engine Requirements.

3.1 Dynamic Elements

Some of the elements in our game engine have dependencies of other elements of our game engine. Because of that, we have a specific initialization order, as seen in figure 3.1, page 8. We also use `StartUp` and `ShutDown` methods instead of constructors and deconstructors, to be able to finely control when and where dependencies are initiated.

This design pattern can also be used to instantiate systems that have circular references, by creating the objects before initializing them. That way both objects can be created, the required parts initialized and then the circular dependent parts can be created.

Our shutdown sequence is the opposite of the startup sequence, to make sure that subsystems are not destroyed before the systems that depend on them.

Physics System

The Physics System controls both movement and collisions in the game world. As the gameplay of most *RTS*-games are in a plane, the physics system is completely *2D*. In order to use it the `physicsModel` components on `Objects` in the scene must be initialized. There are two types of physics models: `Moving` and Static. Thus, the objects must be initialized as such. The following is an example of initializing a `physicsModel` component.

As can be seen the `physicsModel` is also added to the `PhysicsSystem`, which steps every frame checking for collisions between objects in the scene and moving them, if they are moving objects and have assigned a direction. If collisions occurs, a collision event is triggered, which can be subscribed to and handled.

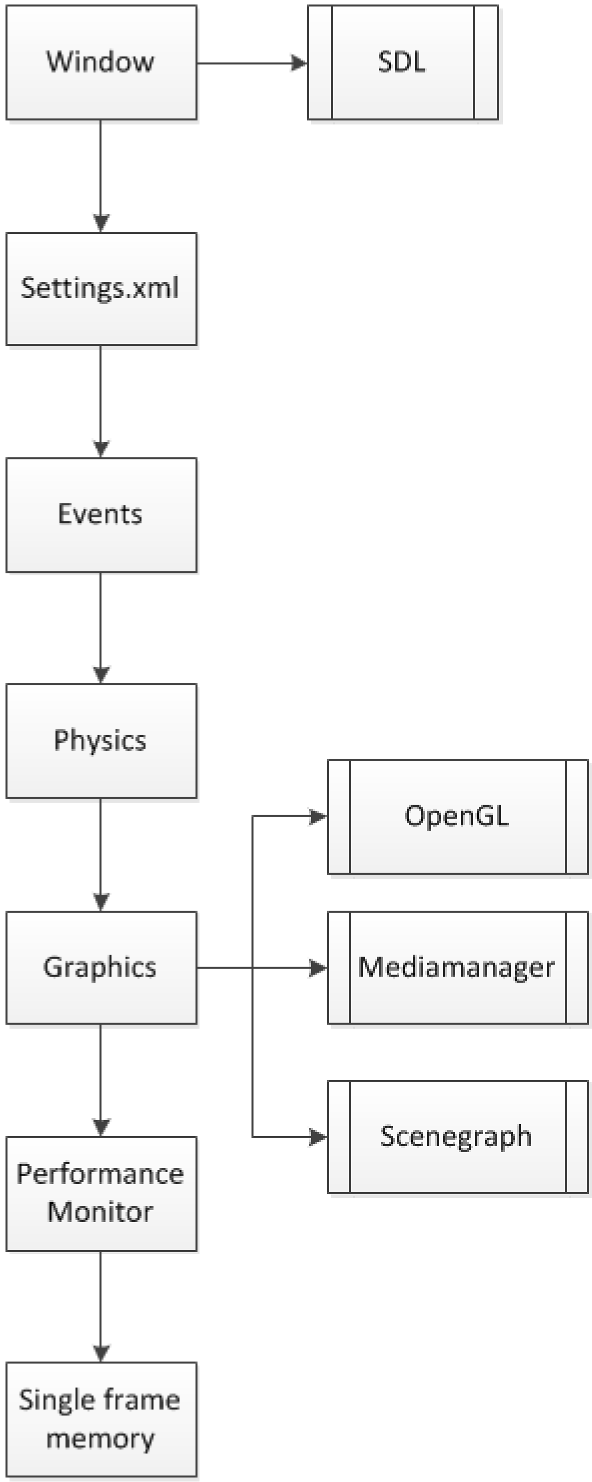

Figure 3.1: Dynamic Elements: Engine startup flowchart.

```
MovingObjectModel* tempMovingObject =
    new MovingObjectModel(CIRCULARSHAPE, PLAYERTYPE, forward, player);
player->physicsModel = tempMovingObject;
Circle circle(Point(0.0f,0.0f),0.5f);
player->physicsModel->InitializeAsCircle(circle);
SINGLETONINSTANCE(PhysicsSystem)->AddMovingObject(tempMovingObject);
```

Figure 3.2: Physics System: Initializing a component.

Path Planner

The `PathPlanner` class can be used in conjunction with the Physics System in order to give moving objects in the scene a target destination that they will travel to using the shortest path without traveling into or through other static objects in the scene. Commonly this would be used to give selected units a move-order by using a mouse click event. But before being able to do so, the map must first be initialized. This is shown in figure 3.3, page 9.

```
SceneGraphManager *createGraph()
{
    float mapWidth = 40.0f;
    auto ground = new Object();
    ground->Name = "Ground";
    ground->model =  SINGLETONINSTANCE( MediaManager )->ground;
    ground->SetPos2D(20.0f,20.0f);
    ground->SetScale(mapWidth,mapWidth,1.0f);
    SINGLETONINSTANCE(PathPlanner)->StartUp(mapWidth);
```

Figure 3.3: Path Planner, 1: Initializing the map.

Firstly, a new object is created in the scene, which will be used as the ground and is assigned the ground model, which essentially is a plane width a height and width of one. This is the scaled to a width and height of 40, or whatever one might want, and positioned so that the lower left corner is in $(0, 0)$. Then the width of the map is parsed to to the singleton instance of the `PathPlanner`.

Secondly, two other objects are added to the scene. These are examples of the two different types of physic objects that are usable. The first object, which is named player is an example of a moving object. It is initialized just like the ground was, but are then added a physics component of a `MovingObjectModel`, which's collider is initialized as a circle. Lastly, the component is added to the list of known moving objects in the `PhysicsSystem`. The exact same thing is done with the box object, but it is just initialized as a `StaticObjectModel` and with a rectangular collider, and is added to the list of known static objects in the `PhysicsSystem`. Note that the player object

```
auto player = new Object();
SINGLETONINSTANCE(PlayerInteraction)->StartUp(player);

player->Name = "Player";
player->model =  SINGLETONINSTANCE( MediaManager )->crazyModel;
MovingObjectModel* tempMovingObject =
    new MovingObjectModel(CIRCULARSHAPE, PLAYERTYPE, forward, player);
player->physicsModel = tempMovingObject;
Circle circle(Point(0.0f,0.0f),0.5f);
player->physicsModel->InitializeAsCircle(circle);
SINGLETONINSTANCE(PhysicsSystem)->AddMovingObject(tempMovingObject);
player->SetPos2D(5,35);
player->Rotate(90.0f, 1.0f, 0.0f, 0.0f);
player->SetForward(0.0f, 1.0f);
player->setLookAt2D(forward.X,forward.Y);

Rectangle physicsBox(Point(-0.5f, -0.5f), 1.0f, 1.0f);

auto box = new Object();
box->Name = "Box";
box->model =  SINGLETONINSTANCE( MediaManager )->boxModel;
StaticObjectModel* tempStaticObject = new StaticObjectModel(RECTANGULARSHAPE);
box->physicsModel = tempStaticObject;
box->physicsModel->InitializeAsRectangle(physicsBox);
SINGLETONINSTANCE(PhysicsSystem)->AddStaticObject(tempStaticObject);
box->SetPos2D(20.0f, 0.0f);
box->SetScale(40.0f, 1.0f, 3.0f);
```

Figure 3.4: Path Planner, 2: Creating the two objects.

also is parsed to the singleton instance of `PlayerInteraction`. This is done in order to move it using the `PathPlanner`, when clicking with the mouse on the ground. In figure 3.4, page 10, it is shown how the these two objects are added.

```
SINGLETONINSTANCE(PhysicsSystem)->SetStaticPathMap();
```

Figure 3.5: Path Planner, 3: Static map path.

Then the path planning map is set in the end of the `createGraph()` function. Once this is done, the `PathPlanner` is ready for use. Note that if any objects are added to or removed from the scene dynamically, then the `SetStaticPathMap()` has to be called again. This is shown in figure 3.5, page 10.

3.2 Resource Management

Handling resources is an essential part of any game engine, and as such it also is a big part of our engine. The handling is done in three different managers: The `MediaManager`, the `MemoryManager`, and the `SettingsManager`.

Media Manager

The *Media Manager* is the tool that is to be used in order to import and access media files for use in the game. Currently, it supports importing of `.TGA`-files as textures and all the common model formats as 3D models, by using *Assimp*[1]. In the future, audio files will be accessed and imported using the media manager as well. As this manager is a singleton, any media imported using it will readily be available anywhere in the code as long as <Managers/MediaManager> is included.

In order to import a resource, one must first create either a model or a texture image and place it in the `Resource` folder. A container must then be created for it in the `MediaManager` class (`Texture*` for textures, and `Model*` for models). Then in the `MediaManager.StartUp()` function the resource must be loaded and stored in the aforementioned container using the `LoadTexture()` or `ImportAssimpModel()` respectively. Hereafter the imported asset can be accessed from the `MediaManger` singleton from anywhere within the engine. However, as the scene is actually built in `SceneData.cpp` and the `createGraph()` function, one would usually access it in here in order to assign it to an object in the scene. The Media Manager is used in the following way:

1. Firstly two containers, to hold the player's model and its texture, are created. See figure 3.6, page 12.
2. Then the resources are imported in the `StartUp()` function. See figure 3.7, page 12.
3. And lastly an object is instantiated in the `createGraph()` function and are assigned the player model that we imported earlier. See figure 3.8, page 12.

One might note that the texture is not assigned here. The reason being that it is done automatically when importing the player model as long as the texture is assigned in the model file is called the same as the texture we imported. In this example the name would have to be `player.tga` or `PlayerTexture`.

[1] See *Assimp* documentation for a full list of supported file formats.

```
class MediaManager
{
    SINGLETON( MediaManager )

public:
    Model* playerModel;
    Texture* playerTexture;
```

Figure 3.6: Media Manager, 1: Initializing containers.

```
void MediaManager::StartUp()
{
    ///IMPORTANT: LOAD TEXTURES BEFORE MODELS
    playerTexture = LoadTexture("Resources/playerTexture.tga", "PlayerTexture");
    playerModel = ImportAssimpModel("Resources/PlayerModel.3ds");
```

Figure 3.7: Media Manager, 2: Resources are imported.

```
SceneGraphManager *createGraph()
{
    Object* player = new Object();
    player->model =  SINGLETONINSTANCE( MediaManager )->playerModel;
```

Figure 3.8: Media Manager, 3: Object instantiated.

Memory Management

Our engine implements a stack based single frame allocator. This is useful for variables, that are created in every frame.

At the beginning of each frame the allocator deletes all allocated memory and therefore each frame starts with a clean slate. It is not possible to deallocate specific elements from the frame allocator. Due to it being stack based, it is only possible to roll back the stack to a specified marked point. This deallocates all memory that was allocated after the marker was retrieved.

We have also implemented a basic smart pointer structure, that can be used, if a general heap allocator wants to be implemented at a later point. The allocator should return a smart pointer (containing the real address to the allocated memory) instead of a normal pointer. This will allow the heap allocator to move the allocated memory blocks around, without breaking the users memory access, if the smart pointers internal pointer is updated with the new memory location. This is necessary because a general memory allocator like a heap allocator can become fragmented. It is then necessary to clean up once in a while, so the allocator does not waste too much space on uninitialized memory blocks that are too small to actually contain anything. If the defragmentation method splits up the defragmentation over several frames, the memory can stay more or less defragmented while preventing excessive CPU usage.

Settings Manager

As every game needs variable settings and options, which should be easy to customize. We decided to keep the data in an external `XML`-file, so that it is easily changeable outside the game. This opens up for the possibility of having a light-weight external tool.

Inside the engine, the `XML`-file is loaded upon startup and stored as a tree[2], which is easily traversable and logically structured. In the same way, the tree can be exported back to the `XML`-file at any given time, and will also do so upon shutting down the engine.

3.3 Graphics System

The Graphics System is in charge of rendering any graphics and light in the scene. It does so using the `Render()` function, which basically sets the camera position and direction every frame, so that movements of the camera actually moves the viewport. Furthermore, it runs recursively through the scenegraph, which is created in the `SceneGraphManager`, and runs the `Render()` function on the models assigned to all objects in the graph. It goes through the root node down through all its children pushing and popping the `glMatrix`, which basically means that objects are placed in the scene relatively to their parent objects.

Camera

The camera is a class that is used to define the viewport; Its position and direction in the world. As such it is part of the scenegraph. The thing to note about the camera is its control scheme. As this engine is an RTS engine, the genre dictates the movement possibilities and they basically consist of moving the camera on a plane parallel to the ground, when moving the cursor to the edges of the viewport.

Lighting Manager

This manager contains a list of eight lights, since that is the amount of lights that *OpenGL* lets you have in the scene at any given time. The light class contains the different color properties of the light, a position, and whether it is in use or not. When setting up a light in the scene, you can simply call the `getAvailableLightIndex()` function in the singleton instance of the

[2] In the "Document Object Model"-standard.

`LightingManager` class. It is then given an index of a free light, if one is available. This can then be used to to access the given light in the `LightingManager` instance, set its color properties, position, and enable it. This way it is easy to set lights in the scene without destroying others. An example of this usage can be seen in figure 3.9, page 14.

```
int lightIndex = SINGLETONINSTANCE(LightingManager)->getAvailableLightIndex();
SINGLETONINSTANCE(LightingManager)->lights[lightIndex].enable(true);
SINGLETONINSTANCE(LightingManager)->lights[lightIndex].setDiffuse(0.5f,0.5f,0.5f, 1.0f);
SINGLETONINSTANCE(LightingManager)->lights[lightIndex].setSpecular(0.7f,0.2f,0.1f, 1.0f);
SINGLETONINSTANCE(LightingManager)->lights[lightIndex].setAmbient(0.2f,0.2f,0.2f, 1.0f);
SINGLETONINSTANCE(LightingManager)->lights[lightIndex].setPos(10.0f,10.0f,10.0f);
```

Figure 3.9: Lighting Manager: Setting up a light.

3.4 Game Loop

The game loop catches and handles all input and game events first. Then, the physics system is run with the elapsed time since the last update. When the 'model' of the world is updated, everything is drawn. In the end, we measure our performance and frame memory usage. A visual representation can be seen in figure 3.10, page 15.

Event System

The general event system is structured with event listeners and event managers as described in *Game Coding Complete*[3]. When a system like the physics system wants to update a moving object it can send an event to the event handler. All subsystems who want information about that event then gets notified. That keeps the relationship between the subsystems to a minimum, since they only have to keep track of themselves and the event manager.

3.5 Input

The engine uses the SDL API to register keyboard and mouse input using their built in event types. When the engine is started a loop runs using the `SDL_PollEvent` method. A switch-case calls the appropriate handlers when SDL events is registered.

[3]"Game Coding Complete", Mike McShaffry et al. 3rd ed. Chapter 10.

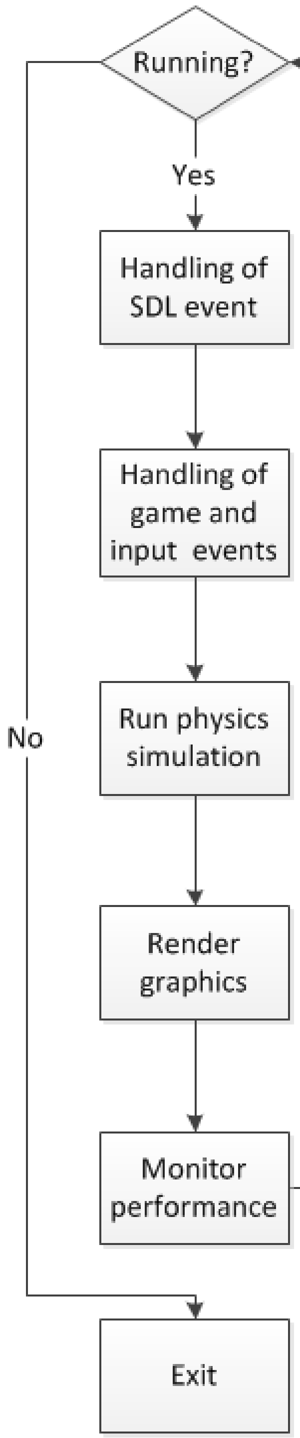

Figure 3.10: Game Loop: Flowchart showing the main game loop.

Mouse

When a mouse-press is registered window coordinates are translated into *OpenGL* world space coordinates and an event for pathfinding and movement of the player is triggered.

The mapping from screen coordinates to world coordinates happen via an inverse projection matrix.

Keyboard

When the keyboard are pressed, an event with the appropriate key is created which other subsystems can listen for. The engine currently only listens for escape key and exits when pressed.

3.6 Advanced Feature: Performance Logging

The performance logger enables the user of the engine to find out which parts of the code are taking the most time. The performance logger allows the user to measure time and save the measurements in distinct, named, lists. E.g. the physics step could be measured and saved in a list named physics. The lists are saved to a `CSV`-file when they are full. We specified that the lists should contain at most `1000` elements, which will correspond to `1000` frames, about 1-2 seconds on our development machines, if a certain section is logged every frame.

We decided against doing the graphing of logged entries ourselves, as an offline component like *Microsoft Excel* excels at this.

Post Mortem

4

Having completed the engine or at least come as far as was possible for us in the time given, it seems feasible to reflect on the process. What went wrong, what went right, and what have we learned from the experience.

4.1 What Went Wrong?

The process of creating this game engine hit a few bumps in the road. The first problem was to just start implementing different features, instead of starting out by thoroughly thinking the basic design structure of the engine through. As a result we fairly early had an engine with a lot of working features, but with a basic structure that did not make a lot of sense. Thus, we tried to go back and redesign the basic structure and then refacture the features already implemented, in order to fit them into the new improved structure. This task, however, was more cumbersome than expected and left us with a broken engine for quite a long while. Nevertheless, the task succeeded in the end.

4.2 What Went Right?

In the end an engine was made, that included all the required features and have the beginnings of a full fletched *RTS* engine. Furthermore, the process was rather educational as to what not to do when creating an engine; that time spent designing the architecture of the engine will be time saved in the long run. Trying to make most of everything from scratch, and not use more libraries than needed, was good practice, as it gave a unique insight into the workings of such features as well as to fully visualize the time that could be saved using third-party libraries.

4.3 Engine Examples

This section will show a few examples, showcasing the various features of engine.

Heads-Up Display Usage

To demonstrate that our *HUD* is interactable, we made it so that clicking the white section of the screen would spawn a box in the playing field. The difference can be seen between figure 4.1, page 18 and figure 4.2, page 19.

Figure 4.1: Engine Examples, HUD Usage 1: A few boxes.

Screen Panning

By moving the mouse cursor to either of the screen's sides, the camera will pan in that direction. In the example shown in figure 4.3, page 20, the camera has panned to the right.

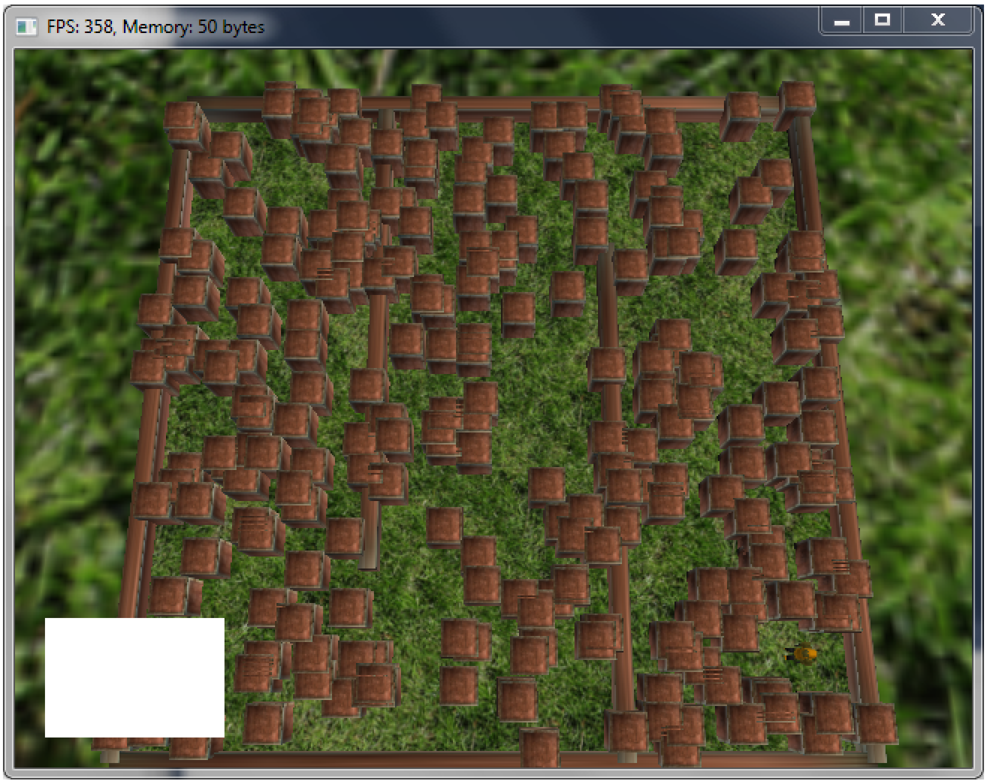

Figure 4.2: Engine Examples, HUD Usage 2: A lot of boxes.

Figure 4.3: Engine Examples, Panning: Camera has panned right.