

Frontpage for written work/project

ITU student <input checked="" type="checkbox"/> Course <input type="checkbox"/> Thesis <input type="checkbox"/> Project (with project agreement)	EBUSS student <input type="checkbox"/> Course <input type="checkbox"/> Thesis <input type="checkbox"/> Project <input type="checkbox"/> Summerproject <input type="checkbox"/> 4-weeks project <input type="checkbox"/> 12-weeks project <input type="checkbox"/> 16-weeks project
--	---

Title of course, project or thesis:

Game Engines

Course manager/Supervisor(s):

Mark Jason Nelson

Name(s):

1. Emil Erik Hansen

2. Julian Møller

3. Mads Johansen

4. René Bech Korsgaard

5. Steen Nordsmark Pedersen

Birthday and year:

14-06-85

18-03-87

22-05-85

28-05-84

15-02-87

ITU-mail:

emha @

jumo @

madj @

reko @

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

Contents	iii
List of Figures	iv
1 Formalities	1
2 Engine Description	2
2.1 Engine Scope	2
2.2 Design Rationales	2
2.3 Major Features	5
3 Implementation Overview	6
3.1 Dynamic Elements	6
3.2 Resource Management	8
3.3 Rendering	10
3.4 Game Loop	10
3.5 Input	10
3.6 Performance Logging	10
4 Examples	11
5 Concluding Comments	12

List of Figures

2.1	Engine Scope: Screenshot from <i>DOTA2</i>	3
3.1	Path Planner, 1: Initializing the map.	6
3.2	Path Planner, 2: Creating the two objects.	7
3.3	Path Planner, 3: Static map path.	8
3.4	Media Manager, 1: Initializing containers.	9
3.5	Media Manager, 2: Resources are imported.	9
3.6	Media Manager, 3: Object instantiated.	9

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 the requirements specified on the course website¹.

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 “Hero RTS¹”-genre, more commonly known as “MOBA²”. This includes games like *Heroes of Newerth* and *League of Legends*, both heavy inspired by the *WarCraft III* modification *Defense of the Ancients*.

To make our engine achieve the requirements of the genre, it was important to have a clear idea of what 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. On top of that, there has to be an interactable HUD³. The camera will do limited movement, which again will limit the amount of required graphics that needs to be rendered. Furthermore, the avatar of the player has to be able to get from A to B, and be able to pathfind around various obstacles.

This puts the main focus of the engine on effective handling of mouse events, collision-detection and pathfinding.

2.2 Design Rationales

This section covers what decisions was put behind the major parts of the engine, namely *Graphics*, *Physics System* and the *Path Planner*.

¹Real-Time Strategy.

²Multiplayer Online Battle Arena.

³Heads Up Display.



Figure 2.1: Engine Scope: Screenshot from DOTA2.

Graphics

It was clear early on that graphics would be an important part of the 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 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 importation of a wide range of file formats. On the other hand creating a custom loader will let the authors 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 today. A custom MS3D-loader was created as well, but for some reason it just would not function, and the authors were forced to utilize a library for model loading. The library of choice was *Assimp*, since it was fairly simple to use and allowed for importation of a lot of 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 working a 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 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.

Physics System

The physics system in an *RTS* is in 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 as such it had to be implemented into our engine. This could be done in a number of different ways such as a navigation mesh, grid based path finding, using an A* algorithm or breadth first etc.. Since the world of *RTS* games are typically fairly open and for simplicity it was chosen to make the path finding grid based. The precision of the pathing could then be altered via the number of grid divisions. However, the precision is reverse 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 set object into the scene and parse the size of the map to the path planner class. Then they would have a functional path finding. With a navigation mesh the user would have to define each node by defining a convex polygon. This is arguable a lot more cumbersome.

As regards to the actual algorithm A* was chosen, 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 Major Features

To make our game function optimally for the specific game genre, the main - and most powerful - features are the following:

- An effective Mouse Event system.
- A HUD that the mouse can interact with.
- Effective pathfinding.

Implementation Overview

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

Physics System

Collision Detection

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 3.1, page 6.

```
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 3.1: Path Planner, 1: Initializing the map.

Firstly, the 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 then 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 the singleton instance of the PathPlanner.

```

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 3.2: Path Planner, 2: Creating the two objects.

Secondly, two other objects are added to the scene. These are examples of the two different types of physic objects that are useable. The first object, which is dubbed 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 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 3.2, page 7, it is shown how these two objects are

added.

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

Figure 3.3: 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 3.3, page 8.

3.2 Resource Management

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 you include `<Managers/MediaManager>`.

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 in here in order to assign it to an object in the scene. The Memory Manager (described in section 3.2, page 8 is used in the following way:

1. Firstly two containers, to hold the player's model and its texture, are created. See figure 3.4, page 9.
2. Then the resources are imported in the `Startup()` function. See figure 3.5, page 9.

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

3. And lastly an object is instantiated in the `createGraph()` function and are assigned the player model that we imported earlier. See figure 3.6, page 9.

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`.

```
class MediaManager
{
    SINGLETON( MediaManager )

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

Figure 3.4: 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 3.5: Media Manager, 2: Resources are imported.

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

Figure 3.6: Media Manager, 3: Object instantiated.

Memory Management

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 changable 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 DOM² Tree, 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 Rendering

Camera

Lighting Manager

3.4 Game Loop

Event System

3.5 Input

Mouse

Keyboard

3.6 Performance Logging

²The “Document Object Model”-standard.

Examples

4

Concluding Comments

5