Ohunkakan

Technical Design Document
Game Engine Programming, 2020

Adam Jóźwiak	216786@edu.p.lodz.pl
Jan Klamka	216797@edu.p.lodz.pl
Piotr Pilecki	216865@edu.p.lodz.pl
Maciej Pracucik	216869@edu.p.lodz.pl
Tomasz Witczak	216920@edu.p.lodz.pl

July 5, 2020

Contents

1 Requirements and features

- procedurally generated endless scene
- conversion and loading Unity scenes
- fire effect simulation
- various post-processing effects
- AABB collision detection and resolution

- multiple point lights
- scene graph with dirty flag
- simple GUI (buttons and text)
- external scene editor
- game logic encapsulated in .dll scripts
- skinned animations
- PBR renderer
- 3D sound
- basic game state machine

2 Engine architecture

2.1 ECS

2.1.1 Overview

The ECS system used in our engine is purely based on components. The entities are only identifiers and components are stored in the appropriate arrays in component manager.

This system is the core of the engine. It allows us to decouple various operations that could be done by putting them in separate systems. The components that we used are loosely based on the ones found in the Unity engine.

The event system is integrated with ECS for the simplicity of usage.

All managers implement a singleton pattern for a simpler API usage and mitigating the access issues.

2.1.2 Entity

The entity itself is just a number, unsigned integer to be precise.

There is a maximum number of entities that can be created and it is defined by the appropriate macro.

Figure 1: ECS architecture and event system

The EntityManager class handles the entity-related operations. It's responsible for creating and destroying entities, as well as accessing the component signature which stores the information about the components that the chosen entity has.

Entities, as stated before, are only integers. Because of that, there is a queue containing all the available identifiers, filled at startup. Then, whenever the user wants to create an entity, the element (identifier) is popped from the queue and returned to the user. Whenever the programmer wants to destroy the entity, the operation is performed in reverse - the identifier gets put back onto the queue.

The signature that stores the information about the component that an entity has is a bitset. When an entity has the specific component, the corresponding bit is set to one. When it doesn't, it's set to zero.

2.1.3 Component

There is no inheritance hierarchy if it comes to components. Each component should be a POD struct. The information that a struct is a component is done with metaprogramming and templates.

Each component has its own identifier which needs to be manually set by the programmer when creating a component class. The identifier is stored as a template specialisation of the member function of the ComponentRegistrant class. In that way, we can have a different return value for the method that depends solely on the type of the component.

After setting the identifier, each component must be registered manually by invoking the registration function of the component manager which creates the array for the components of the chosen type.

There is a custom class that represents the array of components. It's implemented to an interface so that it can be stored and accessed by the manager. The components are stored in a plain simple array of the chosen type. This ensures that there is no dynamic allocation at runtime. All of the memory is being statically allocated which means that that system will be faster than the dynamic one.

The main point of storing the components in a static array is to have it be tightly packed. To achieve that whenever the component is removed from the array, the last element in the array is copied in the place of the deleted one.

The array that stores the components works a bit like a map but with the entity identifier as a key.

2.1.4 System

Each system needs to inherit from the base system class. The base class contains a set of all the entities that fit into the filtering criteria. It also exposes the interface for the system that allows the user to do the setup, cleanup, set the filters and do the update.

Systems work based on filters. A filter is just another component signature but this time it's not used to specify which component an entity has. This time it tells us which entities we want to have in our system container. It filters only these entities that have the specified set of components. It does it by comparing the signatures of the systems and components and then updating the entity sets in systems.

Systems also need to be registered but this time the process is automatic. Each system inherits from the SystemRegistrant template class which has a one static object of the Registrant class. In the constructor of said object, we call the registering function of the system manager. The fact that this object is static tells us that the registration process will happen only once per type.

The registration creates the instance of the appropriate system in the system manager. The systems are stored in a map that takes the type identifier as a key and stores the value of a pointer to the system base class.

2.1.5 Registry

To keep it simple, Registry serves as a facade to all the managers. It wraps all the important operations and provides the user with a simple interface.

With the registry, the users can create and destroy entities, manage addition and deletion of components, set the system filters and also set up the event system. Registry also serves an important function of synchronizing all the managers with the up-to-date signatures of systems and entities.

2.1.6 Utility

For even simpler usage, there exists the Entity class that acts as a thin wrapper around the entity identifier. It enables the user to write in a simpler syntax by omitting the registry and using the methods of the entity itself.

As a result, all the operations that one would want to perform on the entity or its components can be done with the entity object only. The usage of singletons for managers allows for this type of free access, regardless of the place in the program where it's being called.

2.2 Events system

Each event, just like component, can be just a POD type.

First thing that needs to be done when the user wants use the event system is to register the listeners. They can be just free functions or even members of classes. Then with the access to the Registry class, the user can just send the event object through the appropriate send method. The manager then iterates over its listener container and calls the functions that listen to that specific type of an event.

3 Entities

- Terrain
  - Grass
  - Tree
  - Ground
  - Rocks (walls, obstacles, decorations)
- Billboards
  - Flame
  - Enemies
    - Bishop
    - Rook
    - Enemy moving sideways
    - Enemy moving back and forward
    - Enemy moving up and down
- Enemy AI boundaries
- Level colliders

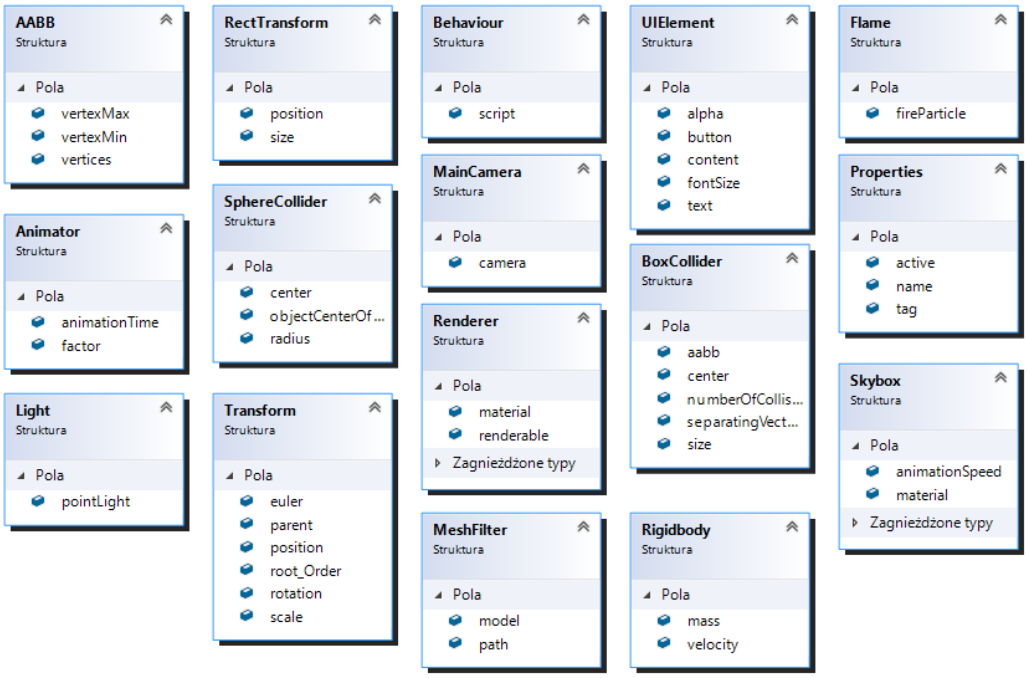

Figure 2: Components

- Traps
- Waterfalls
- UI elements
  - Buttons
  - Text
- Player

4 Components

4.1 AABB

It defines two points in the space corresponding to model's minimum and maximum vertex position values on every axis. It's used for creating the box colliders.

4.2 Animator

It contains the information about the duration and the velocity of the animation.

4.3 Behaviour

It adds the behaviour scripts to entities for defining the object logic.

4.4 BoxCollider

It adds the bounding box for collision detection. Whenever the collider surface intercepts with a different one, collider system sends an event that later can be resolved in a specific behaviour script.

4.5 Flame

It adds an animated fire particle billboard.

4.6 Light

It attaches a point light to the entity.

4.7 MainCamera

It defines the main camera used in the game.

4.8 MeshFilter

It adds a model to the entity and contains the information about the file path in the project files.

4.9 Properties

It contains the information about the name and the tag of the entity. It also defines whether it is active in the scene.

4.10 RectTransform

It defines the position in the scene and the size of the UI element.

4.11 Renderer

It contains the information about the material properties, such as paths to the appropriate textures and heights used in parallax mapping.

4.12 Rigidbody

It defines the mass and the velocity of the entity and decides whether it is going to be affected by the physics system.

4.13 Skybox

It contains the information about the textures that create a cube map and the animation speed defining how quickly the skybox will be animated.

4.14 SphereCollider

It adds a bounding sphere that's used for detecting the collision by checking whether the distance between the colliding object and the center of the sphere collider is smaller than the value of the sphere radius.

4.15 Transform

It contains the information about the position of the object, the rotation and the scale in the world.

4.16 UIElement

It defines whether or not the entity is a UI element and what type of an element it is, e.g. text or button.

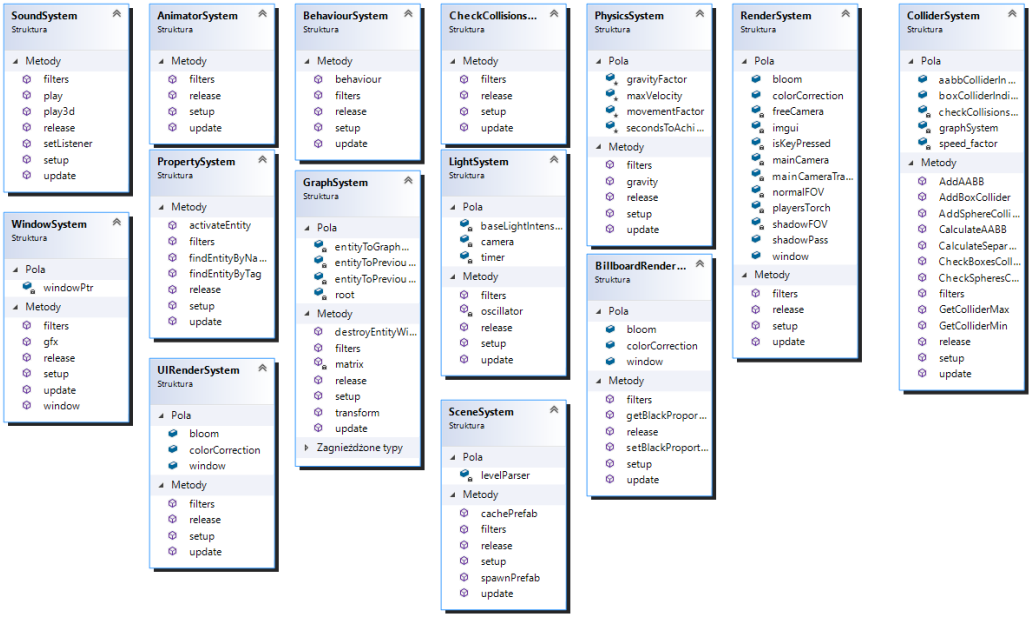

Figure 3: Systems

5 Systems

5.1 Animator

This system is responsible for advancing the animation times for these models which should be animated.

5.2 Behaviour

It's a system for handling behaviour scripts, e.g. CameraController. It runs their updates but also loads the appropriate *.dll*'s at the game startup.

5.3 BillboardRender

This system is responsible for rendering billboards, i.e. flames and enemies. It also sets and updates their positions in the world. The system contains the last post-processing passes and because of that it exposes the interface to set the black fade effect in the color correction post-process.

5.4 CheckCollision

The sole purpose of this system is to act as a filter for the entities that should be taken into account when checking collisions. The filtered objects are then used in the ColliderSystem.

5.5 Collider

It's a system which exposes specific methods for handling all things related to collisions. It enables creating and updating the colliders (boxes and spheres) as well as checking and resolving collisions. It is also used for frustum culling with its AABB functionality.

5.6 Graph

This system implements the scene graph with the dirty flag. It maintains a graph structure of all the entities and checks whether or not its transform or activity status have changed since the last frame. In that case, the transform matrices are recalculated. The system also exposes the interface for all the other systems to access the cascaded model transformation matrix that includes the hierarchy between objects. Whats more, it also has a method for deleting entities with respect to that hierarchy.

5.7 Light

It handles updates related to lights in the scene. It is used for setting and updating their positions as well as adding an oscillatory flicker effect to their intensity level.

5.8 Physics

The purpose of this system is to handle the gravity for the objects that need it.

5.9 Property

Although it's a system, it doesn't function as one. The only purpose of it is to expose the interface for looking for entities by their name or tag.

Furthermore, it has a method for enabling or disabling the entity but, on the contrary to the graph system, it bypasses the hierarchy of objects and affects the entity only.

5.10 Render

It is responsible for rendering all entities in the scene. It performs frustum culling so only the entities that are inside the viewing frustum are rendered. It also renders point shadows and starts applying the post-processing effects such as bloom. The next part of the rendering pipeline is then continued in the billboard rendering system.

5.11 Scene

It's a system for loading the startup scene. It is also responsible for spawning prefabs or caching them which is useful when applied to world chunks.

5.12 Sound

This system handles the sound effects. It loads and caches their files at startup. Every frame it updates the 3D sound source positions. Moreover, the system exposes a few basic methods for playing sounds that could be used for example in the scripts.

5.13 UIRender

It's the last system in the rendering pipeline of the engine. Following after the billboard system, its responsibility lies in rendering every UI element in the scene, e.g. buttons or text. It also ends the frame render which performs the buffer swap.

5.14 Window

This system creates the game window, sets its resolution and the name but also makes the window pointer available for the other systems to use.

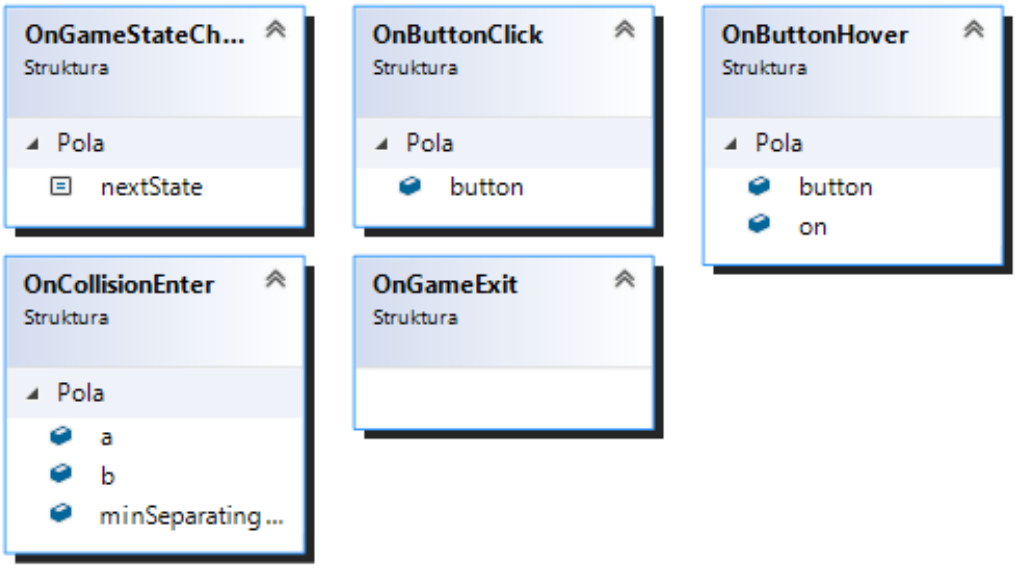

Figure 4: Events

6 Events

6.1 OnButtonClick

After a button is clicked on with the mouse, this event is being sent with the information about the button.

6.2 OnButtonHover

It is sent whenever the player hovers over a button using the mouse.

6.3 OnCollisionEnter

In case collider system detects a collision, it sends this event so that later it can be reacted to by the programmer e.g. in the behaviour scripts.

6.4 OnGameExit

After the player decides to exit the game, this event is being sent to the main engine script responsible for the main gameplay logic. What follows after is

the release of all the systems and then the game exits.

6.5 OnGameStateChange

Whenever one of the predefined game states has changed (for example when player clicks the "Play" button in the menu to transition into the game), this event is responsible for informing multiple systems in which state the game is currently in.

7 Scripts

7.1 CameraController

It's responsible for controlling the camera behaviour, switching its position depending on the game state, following the player while playing, shaking the camera to imitate the effect of the cave collapsing.

7.2 EnemyController

It's responsible for controlling the enemies, their movement and the way of interacting with the environment.

7.3 GameManager

It's responsible for the main gameplay logic, i.e. generating the level procedurally, spawning the enemies, the torches, maintaining the proper UI state, switching the game states when needed, etc.

7.4 PlayerController

It's responsible for controlling the player, his movement, switching of the forms, interacting with traps, waterfalls, enemies and torches. It also defines the light emitted by the player and its behaviour.

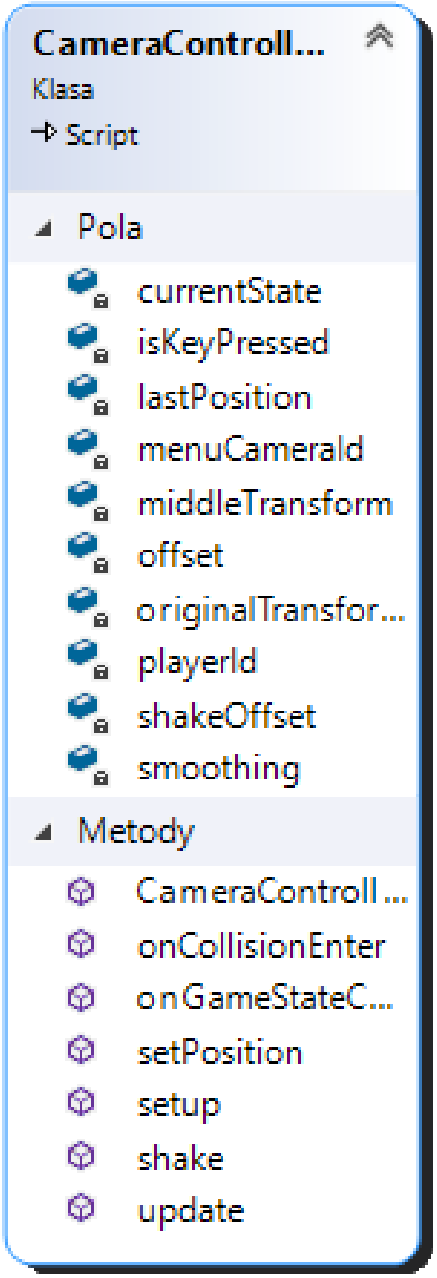

Figure 5: CameraController script

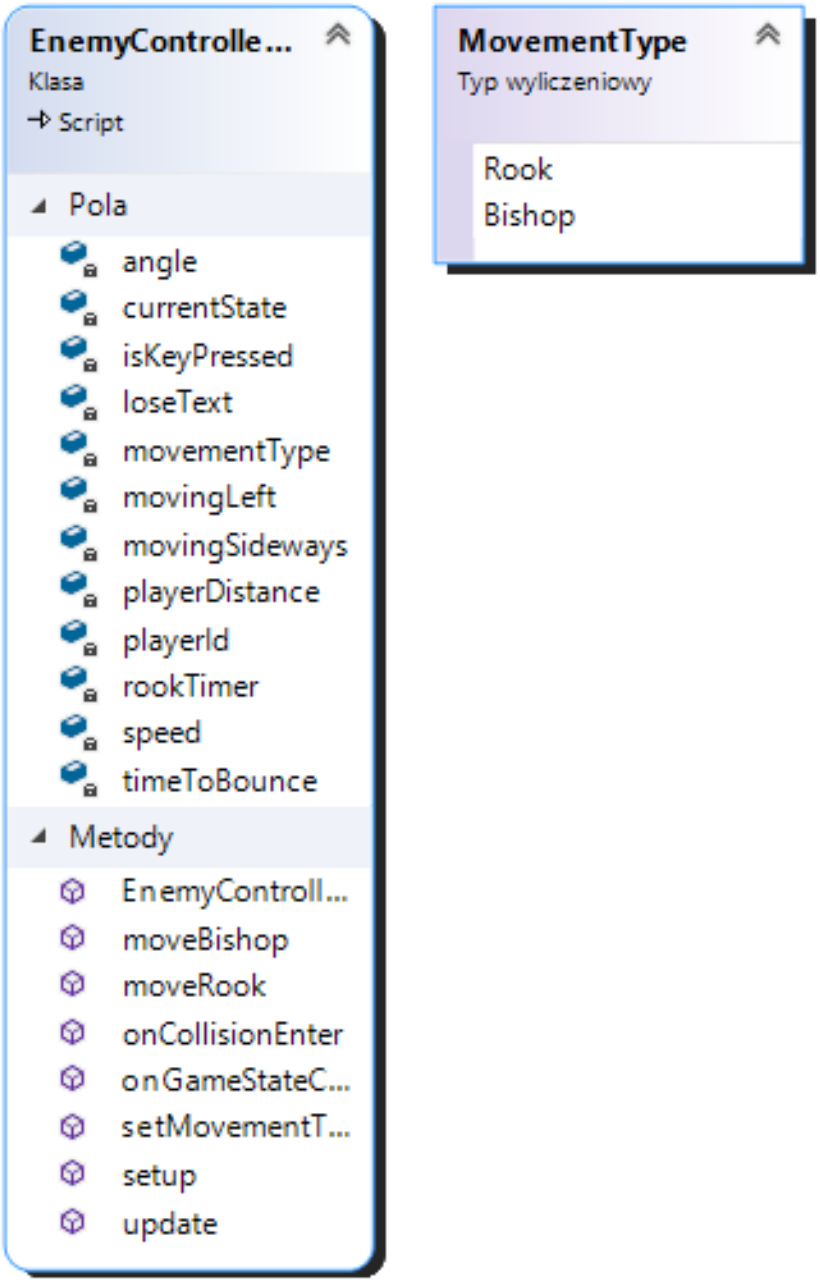

Figure 6: EnemyController script

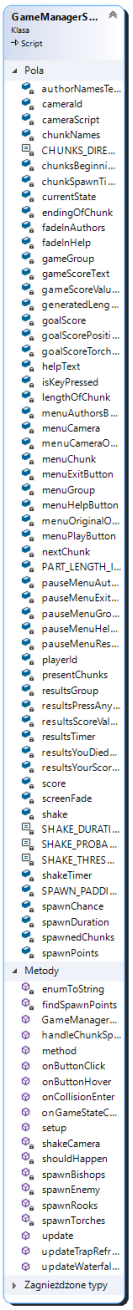

Figure 7: GameManager script

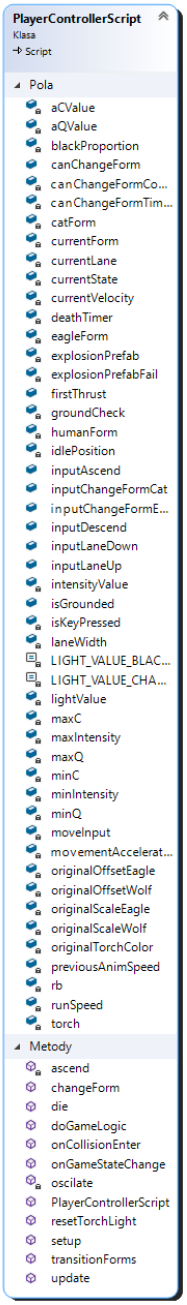

Figure 8: PlayerController script

8 Tools

8.1 Unity

- Setting up the main scene
- Creating the UI
- Creating the layout of game levels as predefined chunks
- Preparing the assets, e.g. textures, models, materials, fonts, etc.
- Storing chunk properties for the in-game procedural generation mechanism
- Creating the prefabs for further processing
- Tagging the objects

8.2 Utility conversion script

- Converting Unity files (such as: .scene, .prefab, .mat, .meta) to our format
- Performing batch conversions
- Copying the asset files around the repository

8.3 Assimp

- Loading 3D models (mainly .fbx, .dae, .gltf formats)
- Loading the necessary information for skinning, e.g. bone identifiers and weights, animation names and durations

8.4 SoLoud

- Loading the sounds (mainly in .wav format)

8.5 ImGUI

- Assisting in the debugging process

8.6 3ds Max / Blender

- Modeling the characters
- Texturing the meshes
- Setting up the bones for the models
- Skinning
- Animating
- Converting between formats

8.7 Photoshop

- Creating and processing textures for PBR materials and other uses

9 Rendering techniques

9.1 Physically Based Rendering

PBR is a technique for real-time realistic lighting. It's based on the information contained in these sets of textures: albedo, ambient occlusion, metalness and roughness. They all keep information that is then used to calculate the radiance of the light. We use the Cook-Torrace BRDF function, the Fresnel-Schilck operation and Smith's geometry method. What makes this physically based is the fact, that the calculations are based on the BRDF function, they use the model of microfacets and they obey the laws of energy conservation. Overall, the results are far better than the ones achieved with the simpler Lambert-Phong models.

9.2 Normal mapping

It's a technique for extending the usage of normal vectors for even better results with the lighting.

The normal and tangent vectors are passed for every vertex of the model. Then in the pixel shader we construct the TBN matrix which is used to create a transformation from tangent to the world space. The normal vector is then read from the normal map texture, properly converted to the [-1,1] range

of values and then transformed to the world space to be used with lighting algorithms.

The TBN matrix is constructed by putting the normal, tangent and bitangent vectors as columns of the matrix. The vectors then serve as the base of the new affine subspace.

What's interesting is that if these vectors are normalized then the resulting TBN matrix is orthogonal which can be used when wanting to find the inverse of the matrix. We can then just calculate it by doing the transposition.

There's a big leap in terms of quality of the rendered models when using normal mapping, yet it's just a slight illusion which breaks when looking at object from an angle. We then see that the geometry is not changed but nonetheless the effect provides us with great results.

9.3 Parallax mapping

The variant of the algorithm that we used here is called Parallax Occlusion Mapping.

Firstly, we calculate the maximal offset of the step we are going to be taking with our texture coordinates. Then we can calculate the number of samples taken depending on the angle of viewing - the larger the angle the more samples we're going to be using.

The whole premise of the algorithm is simple. We take the height/depth map and after transforming the view vector into the tangent space we can perform a stepwise sampling along the direction of the view. We do it in a loop. Take a step. Each step taken is translated into a step in the UV space. Then sample the depth map at that point. Check if we already "crossed" the surface. If not, keep going deeper. Otherwise, interpolate the last two steps and return the new texture coordinate for the parallax-mapped surface.

The effect is best used on plain surfaces. We used it on the ground tiles for the extra level of depth (contrary to normal mapping, parallax simulates the displacement of the geometry).

9.4 Billboarding

To render a billboard, the position of one vertex in the world has to be defined. Then its position is being passed on to the geometry shader, that reads texture dimensions and then creates three additional vertices from which the

billboard will be created; also taking into the account the position of the main camera, so that the image always faces it.

9.5 Fire effect

It is a billboard containing one of the following: albedo texture, noise texture, gradient texture and a mask texture. For a more randomized output look of the flame, the noise texture is used three times with different x and y offsets. Both albedo and mask textures are being transformed over time to create the effect of a "wobbling" fire. Depending on how much of a specific red, green or blue color is needed to be highlighted, three different values may be adjusted to get a different looking flame. In the end the flame is clipped to the shape defined by the mask texture.

9.6 Enemy effect

The effect of the enemy is created in a very similar manner to the fire effect, with slight differences in pixel shader code regarding the offsets to a different set of textures and values multiplying the "animation" speed. Unlike the fire effect, here the mask is not being transformed which gives us a sturdy image with animated insides.

9.7 Skinning

The model for animations is loaded via Assimp library – all the meshes and nodes loaded by it are converted into classes. Assimp also loads the necessary information for animating the model, such as bones identifiers and weights. When the animation is selected, bones are being interpolated between each keyframe, creating the desired animation effect.

9.8 Bloom

The renderer creates two render targets for the scene. One of them needs to be saturated because we want to extract the bright objects. Then we use Gaussian blur on that texture. The next step is adjusting the saturation using the linear interpolation. After this the texture with bloom is combined with the original color.

9.9 Color correction

The used algorithms for color correction are brightness and contrast adjustments, gamma correction and curves based on the input LUT texture.

9.10 Frustum culling

To create the frustum we need six planes. Before each frame every entity in scene graph needs to be checked if it should be rendered. To be shown on the screen, the entity has to be inside the frustum or collide with it (for collision detection we use the AABB bounding boxes).

9.11 Shadow mapping (point lights)

The cube map is first created as a set of six render targets. In the shadow pass, we render the scene to each of the directions of the cube map. In that rendering pass, we calculate the depth of the fragment. Then, in the lighting pass, we can use the information stored in the depth cube map to compare the actual depth of the rendered fragment and decide whether or not the fragment should be lit or covered in shadow.