



IMPORTANT

Before you start, make sure to set the screen resolution of your screen in the settings.ini file located at bin/assets/settings/settings.ini! You can also disable animations and toggle fullscreen for a faster (but less immersive) experience.

Brief Description of Requirements

3D Geometry

We load all geometry using the Model class. The class loads all .obj files and textures. We are using the Assimp model loading library. We store assimp's data in the Mesh class. This class stores the vertex normal, position and texture coordinate vectors for each drawable entity. The Model class creates an object using the loaded meshes. This is done by looping over each mesh, translate each "Assimp Mesh" to our own mesh object and store them in our own object. Furthermore we load the material of the mesh. For better performance, we cache Textures in the storage so we can use them again without reloading.

Geometry mainly created through the GameElements class, which acts as a factory.

Playable

When starting, the framework loads all saveable parameters (see details below). This includes the current level. The level consists of platforms, walls, the goal, power ups and the character. All game elements are instantiated before the game loop.

The player can control the character. The player's aim is to reach the goal before sucked in by the void. The void moves upwards at a steady speed, which can be set in the level settings.

The path which the player might take is blocked at some points so the camera has to be flipped.

When the player wins or loses, the game shuts down (this can be disabled in the settings).

[Tip] You can finish/win the game by jumping upwards, using Space. Rotate the camera by pressing Tab to see behind the walls. Set the flag "debug = true" to disable losing.

Advanced Gameplay

The player has some limitations for reaching the goal. One of them is the partly blocked view (see above), another one is the jump mechanic. The player is only able to jump above a platform (using raycasting, see section *Advanced Physics*). Double jumping is also added to the game, making it harder to reach the goal.

To achieve the required game play features, we implemented the GameManager class which acts as a state machine.

Min. 60 FPS and Framerate Independence

The game runs on (at least) 60 FPS and frame rate independence is reached through delta-timing in the game loop. The physics and game loop run on different time steps.

Win/Loose Condition

The player wins if the goal is reached before the Character touches the void plane. If the plane touches the player first, the game is lost.

In both cases, the game shuts down. This can be disabled in the settings.

Intuitive Controls

The Character is controlled using W, A, S, D and Space. The camera is controlled using Tab. These industry-standard controls seem to be quite intuitive.

Input is fetched using key- and mouse-callbacks.

Illumination Model

The Illumination Model of our game is based on the Bloom effect, which creates the apocalyptic dark vibe. Otherwise we are using simple material.

Textures

As described in “3D Geometry” we use external Textures with texture coordinates.

Moving Objects

The Character moves by input from the player. We achieve this through adding forces. For other movement, refer to *Advanced Physics* and *Vertex Shader Animation*.

Documentation

Quite meta to write a documentation about the documentation, isn't it?

Adjustable Parameters

You are able to adjust nearly everything you can see in the game. For adding a new level, just copy the template file and adjust parameters, make sure to name the file “levelX.ini”. Use the settings.ini to set which level should be loaded (by setting the number of the “level” field, must be of type int). You can also find all settings in this file.

If you want to change the size of all game elements, you can do so in this file. This is neat for debugging and testing.

Another important feature in the settings.ini is the debug-boolean. If set to true, you are able to continue playing after the game is won or lost. You can also use F3 to switch between game- and debug-camera when in debug mode.

Note that you cannot change settings at runtime.

It is also possible to change the screen resolution, turn on/off fullscreen mode and change the refresh rate in the settings.ini. You can also change the brightness (affects Bloom).

If you find reference loading errors, try changing the “assetPath” in settings.ini to your asset path. The settings.ini file must be located in bin/assets/settings!

Note that the assetPath parameter must not contain quotation marks!

Features

Collision Detection

The collision detection is implemented by inheriting the GameManager class from the PxSimulationEventCallback class.

The Character has two separate PxShapes. One for simulation fetching and one for collision triggering.

The collision detection is used for the Power Ups. The Goal doesn't use collision detection as we want the Character to reach the goal fully, not just by touching it.

The GameManager class fetches the onTrigger results and handles the events. Most relevant classes in the project in regards of collision detection are PowerUp, Character and Physics. The setup for the collision detection happens in the init()-method of the Physics class.

Note that when collecting the power up (left above player start position), the void's height decreases by 3.

Advanced Physics

We use physics for moving the player from platform to platform. Each Platform has a static rigid collider while the Character has a dynamic one. The Character is moved by force (using `PxForceMode::eIMPULSE` mode). The Character uses Raycasting to tell if a jump is possible. When pressing Space, the Character requests a raycast downwards (`[0, -1, 0]`) and if the distance is below 10 units, the input is processed. There is a double-jump mechanic which allows to jump twice if received fast enough.

The impact forces depend on the gravity so that the movement is gravity-independent.

Platform, Physics, GameElements, Character are the most relevant classes, as these contain more or less the entire logic for the physics framework.

Update since feedback-talk 1: In addition, when the player reaches the goal, a confetti bomb explodes using physics objects.

HUD

The HUD is implemented so that it shows the most important statistics and data. The following things are shown:

- Level number and level name
- The remaining distance from the Character to the Goal
- The height of the VoidHandler

Implementation is done in the HUD class. To achieve text rendering, we use a library called "FreeType". This is particularly useful, because we can use own fonts. This is done by loading the character glyph, generating the texture and storing it for later usage for all ASCII characters. Because the HUD is independent of the rest of the scene, we use an additional shader.

View Frustum Culling

The View Frustum Culling is used to detect which platforms are visible for the camera and draws them. Platforms which are not visible for the camera will not be drawn.

To check if a platform is inside the camera frustum, we create a testing sphere for every element and check in every frame if the sphere is currently inside the camera frustum. The camera frustum contains six planes (Near, Far, Top, Bottom, Left, Right).

Generating the frustum and additional functions are inside the Frustum class. Every Platform element is checked in the render loop of the main class.

Effects

Lightmaps

Unfortunately, we didn't make it so far, that's why they're missing!

GPU Particle System

The Particle System is implemented using a compute and a geometry shader. We implemented the system based on the provided tutorials. Particles are emitted from the jetback of the character when the player jumps. Each time, 100 particles are emitted but this could easily be much more due to the GPU performance. The system is implemented in the *ParticleSystem.h* file.

Vertex Shader Animation

The meteoroids at the beginning of the level are moved by the vertex shader. This happens in *animated.vs*, where a simple sinus multiplication makes them swing from left to right and vice versa.

For the final submission we implemented a flag, waving in the outer-space-wind. This happens in the vertex shader and now all vertices are moved separate and independently from each other in order to fulfill the provided feedback. The meteoroids were removed completely because they were annoying and didn't add to the experience. Enjoy! (Flag is next to the goal.)

Video Texture

You can toggle video textures on and off by setting *animations* in *settings.ini*.

Basically, every platform, both "horizons", the void and the goal area are animated. This is done by stepping through the different loaded textures each frame. Please note that we decided to only have the goal texture meet the official frame rate and runtime requirements (3 seconds @ 24FPS) to keep the loading time at least at the level it is now. (Note: It's a repeating pattern but it is 3 seconds long.) It would be no problem to just add more images to the textures to increase the FPS but loading takes already very long.

We load the video textures straight forward as .gif files into vectors of unsigned ints and loop through them.

Bloom/Glow

We are using Bloom for the character to illuminate the surroundings (like platforms etc.). This is done by using HDR and extracting bright colors. Furthermore we are using ping-pong framebuffers.

Additional Libraries

Assimp for Model Loading (<https://www.assimp.org/>)

FreeType for HUD (<https://www.freetype.org/>)

PhysX (<https://www.geforce.com/hardware/technology/physx>)

References:

<https://www.freetype.org>

<https://www.informatik-forum.at>

<https://learnopengl.com/>

<http://www.lighthouse3d.com/>

https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Text_Rendering_01#The_FreeType_library

[Tutorials provided in TUWEL](#)

[Hackerman-How-To-Win-Fast-Instructions]

Use every platform until you have to flip the camera for the first time. Then you can skip the platform far left and jump directly to the platform below the goal. Then jump on the goal. If nothing happens, try jumping around a bit and move directly to the center. If still nothing happens, check if you haven't lost the game already. If you haven't, file a bug report.

If you are tired of endless attempts and still cannot make it, change the void speed, the gravity or whatever floats your boat in the settings! ;-)