# ACOMALITH CONFLICT

Lukas Fischer, 1527007
Christian Clemenz, 1226279

## Gameplay


In our game you have to overcome obstacles to reach the end of the level and defeat the evil Dr. Acomalith. The gameplay consists of two main mechanics. The first one is a platform that can be generated under the player's feet if in midair. The second feature is a ball you can throw. Pressing the left mouse button after you threw the ball will teleport you to its current location. The teleportation is used to overcome challenges that can't be solved by jumping. At the end of every level you have to throw the ball at the evil Dr. Acomalith to finish and advance to the next level. If you fall into the lava or hit the spikes you will have to start over.

## Levels

When the game first starts, the first level is loaded. The player has to reach the end of the course and hit Acomalith with the ball to advance to the next level. Alternatively levels can be switched with the buttons 1-4. If you want to start from a certain point of the level you can fly there with the free fly cam.

1. The first level is a tutorial level. Signs explain the basic mechanics of the game.
2. In this level the player has to use the platform the the teleport sphere to scale a tower and avoid spikes and lava.
3. Here you must avoid obstacles and use the teleport sphere and and platform in different ways.
4. The purpose of this level is to show the requirements for this course and to take a closer look at the effects.

# Controls

| | |
|---|---|
| W,A,S,D | Move forward, backward, left, right |
| Right mouse | Pick up teleport ball |
| Left mouse | Throw ball if currently held, teleport to the sphere it if thrown |
| Space bar | Jump if in first person, generate platform if in midair |
| Move mouse in any direction | Change viewing direction |
| R | Reset position of the player |
| M | Mute music and sounds |
| Escape | Exit the game |
| 1-4 | Change level |
| F1 | Open help |
| F2 | Enable/disable debug output on console |
| F3 | Enable/disable wireframe |
| F4 | Switch Texture-Sampling-Quality: Nearest Neighbor/Bilinear |
| F5 | Switch Mip Mapping-Quality: Off/Nearest Neighbor/Linear |
| F6 | Enable/disable normal mapping |
| F7 | Switch between SSAO modes (on, off, ssao only) |
| F8 | Disable/enable view frustum culling |
| F9 | Enable/disable blending (platform) |
| F10 | Switch camera between first person cam and free fly cam (used for debugging) |
| Q, E | Move up or down (only in free fly mode) |
| Shift | Fly faster (only in free fly mode) |
| F11 | Enable bullet debug drawing |

# Requirements

## Freely movable camera

When you start the game, you can move around from a first-person perspective. You can move, jump and use the teleport sphere. Pressing F10 changes to a free fly cam, we primarily use for debugging. This way you can take a better look at the lighting.

## Moving objects

There are already multiple moving objects in the game. The player, the objects that are manipulated by bullet (e.g. teleport sphere, …) and some lights that move around a loaded model.

## Texture Mapping

We load the textures with SOIL and the models with Assimp and the necessary info for rendering are given to the according mesh object. We also use a shader for debugging that isn't affected by the lighting.

## Simple lighting and materials

We have implemented simple point lights with Lambert-Phong illumination. We don't use specular highlights for our game because we want to keep our art style simple. Depending on the level there are some static point lights (mostly where torches are on the wall) and exactly one light, that is casting the shadows.
The first level has shadows right at the beginning when you start. In the second level the shadow caster is rotating light halfway to the top of the tower. In the third the shadow caster is located near the end of the level.

## Complex objects

Most objects are simple in terms of geometry, but we implemented our game so that it also works with more complex objects. In the fourth level you can take a look at a testobject with many vertices that shows lighting and shadows. With the free fly cam you can also take a closer look at the normal maps.

## Animated objects

To show the combination of model matrices, we rotate multiple little spheres around the teleport sphere.

## View frustum culling

For the view frustum culling we calculate a bounding sphere for each object and during runtime each object (only those which aren't moving) gets checked if it is inside the view frustum of the player. You can see the number of tris drawn in the debug output of the console.

## Experimenting with OpenGL

Mip Mapping and Texture-Sampling-Quality settings can be switched with the F4 and F5 buttons. Blending can be enabled with with F9 and can be seen on the plane that is generated with space bar.

## Effects

**Normal mapping**
We use normal mapping on almost all objects. For that we calculate the object's tangents when loading the model and use the Tangent Bitangent Normal Matrix to get the normal vector from a texture (normal map).
References:

- https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping
- OpenGL Programming Guide Ninth Edition (Big red book)

**Shadows**
For shadows we use Shadow Mapping. Since we only have point lights we need omnidirectional shadow mapping. The scene gets rendered once through special shaders to write the depth of each vertex into a texture contained in a cube map. The cubemap is then used for the shadow caster to calculate if a fragment is light by it or not. To smooth the achieved shadows we use PCF by sampling through the neighbor texels in the cubemap and averaging them. To save some performance
References:

- https://learnopengl.com/#!Advanced-Lighting/Shadows/Point-Shadows
- http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

Both of these techniques were combined in one shader (both in normalMap and lambertPhong)

**Screen space ambient occlusion**
First the image gets rendered into a framebuffer with calculated light intensities and shadows. Then the content of the framebuffer is rendered into another framebuffer as a screen space plane with SSAO values calculated. Finally the two framebuffers are combined in a shader (ssao gets blurred) and the result is presented as another screen space plane. Number of samples, radius and bias can be configured via the config file.
References:

- https://learnopengl.com/#!Advanced-Lighting/SSAO
- https://mtnphil.wordpress.com/2013/06/26/know-your-ssao-artifacts/

## Configuration

In the configuration file config.ini you can adjust some parameters:

- screen_width, screen_height
  With these you can change the resolution.
- fullscreen
  You can make the game full screen if it is set to true.
- mouse_sensitivity
  This changes the speed at which the camera rotates when looking around.
- refresh_rate
  This changes the refresh rate of the frames.
- printFPS_in_console, printFPS_every_x_frames
  These enable/disable the debug output (fps, frame time and tris drawn) and how frequent it gets printed.
- ssao_samples, ssao_radius, ssao_bias
  With these you can change how the ssao effect looks.
- ambient
  This parameter changes the overall brightness of the game.

## Libraries

- Bullet physics
  http://bulletphysics.org/wordpress/
- Assimp for loading the models
  http://assimp.sourceforge.net/
- SOIL for loading images
  http://www.lonesock.net/soil.html
- OpenGL, GLFW, GLEW, GLM
- Spdlog a c++ logging library
  https://github.com/gabime/spdlog
- RapidXml - C++ xml parser
  http://rapidxml.sourceforge.net/

All our models were built in blender and exported to obj files. Normal textures were created with Crazybump http://www.crazybump.com/.

Our levels are loaded from xml files. The model were assembled to levels in unity, for which we wrote an extension to export the properties of the models (position, rotation, scale, properties, …) as an xml file.