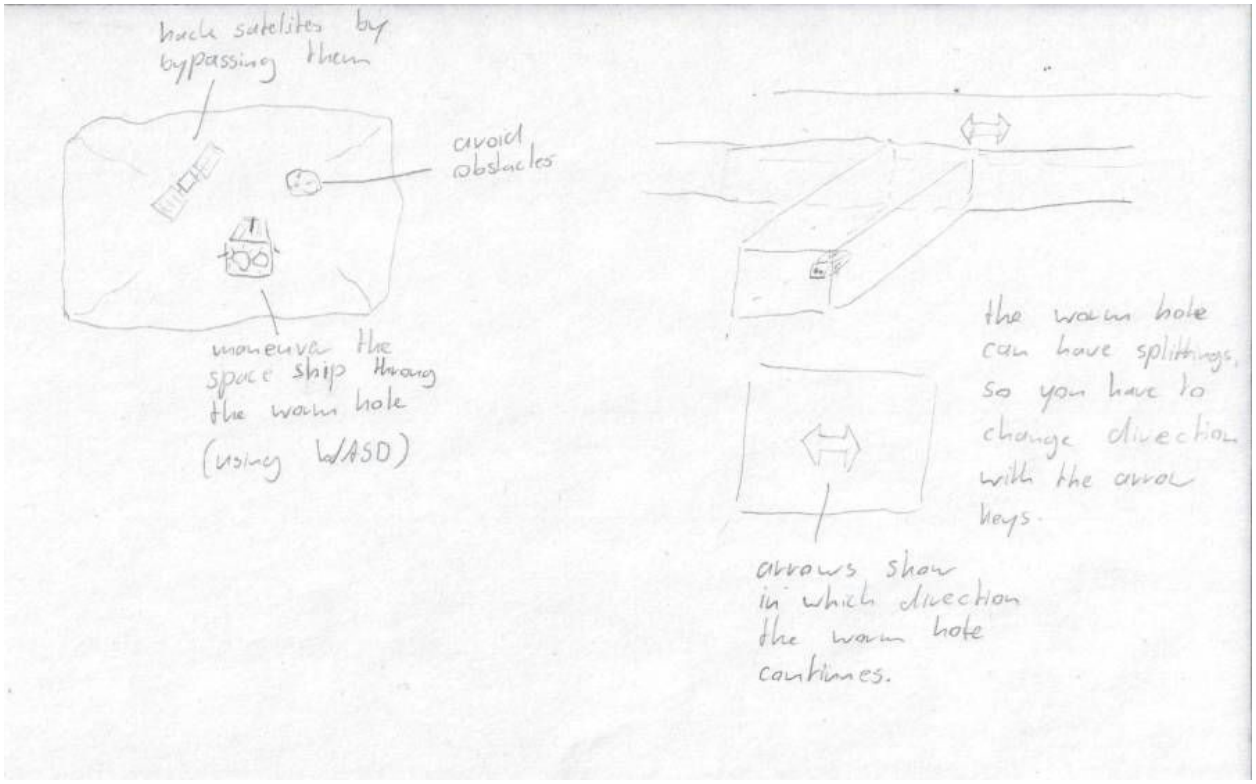
Documentation AnonSpace
Second Submission, 22.6.2015
Wen Chao Chen 1129468
Johannes Dostal 1226714

Description

“AnonSpace” is a running game in wormholes. You control a spaceship of the anonymous imperial galactic colony which has been stuck in a wormhole and your task is to maneuver the ship through the wormhole while hacking satellites and avoiding obstacles.

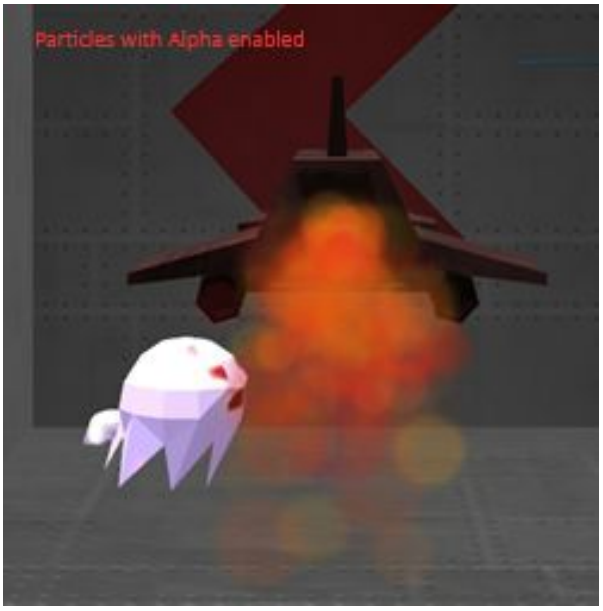
Gameplay

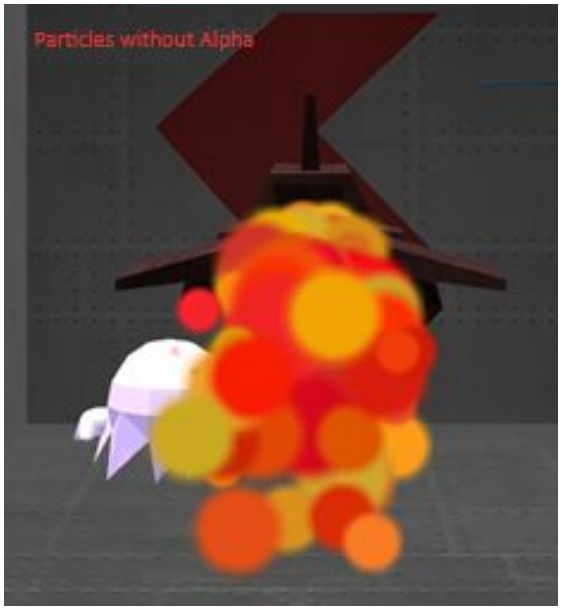
The player controls a spaceship and has to hack as many satellites as possible during his run in the worm hole. The purpose is to escape the wormhole while staying alive by preventing the ship from crashing into the borders of the worm hole or into any obstacle. You can raise your score by bypassing satellites. By reaching a certain level in a given time frame, you win the game and the ship may escape the worm hole. Otherwise, you lose the game.
You have 15 seconds before you die, each collected satellite gives you additional 3 seconds to survive. You have so reach 1000 points, in order to escape the wormhole and win the game. For each second surviving in the wormhole, you earn one point. Each collected satellite gives you additional 75 points.



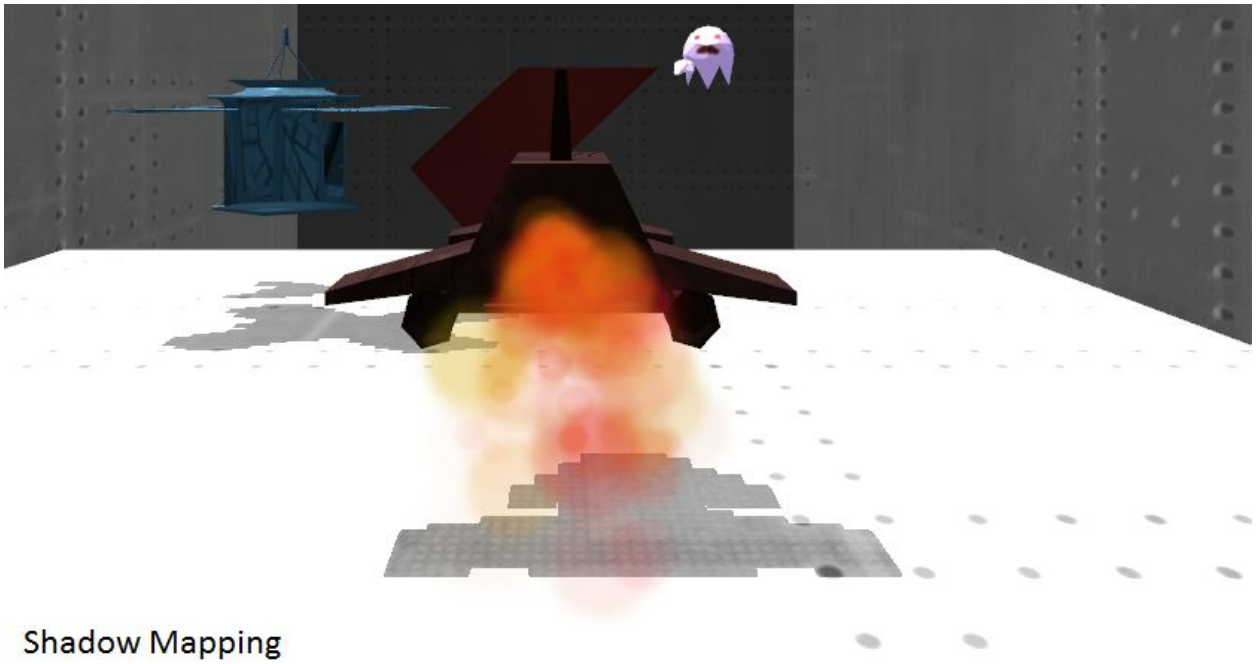
Effects

- **CPU Particle System (+Instancing)**: is implemented in ParticleManager.cpp following the instructions of
  http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing
  You can see the particles as the trail of the spaceship



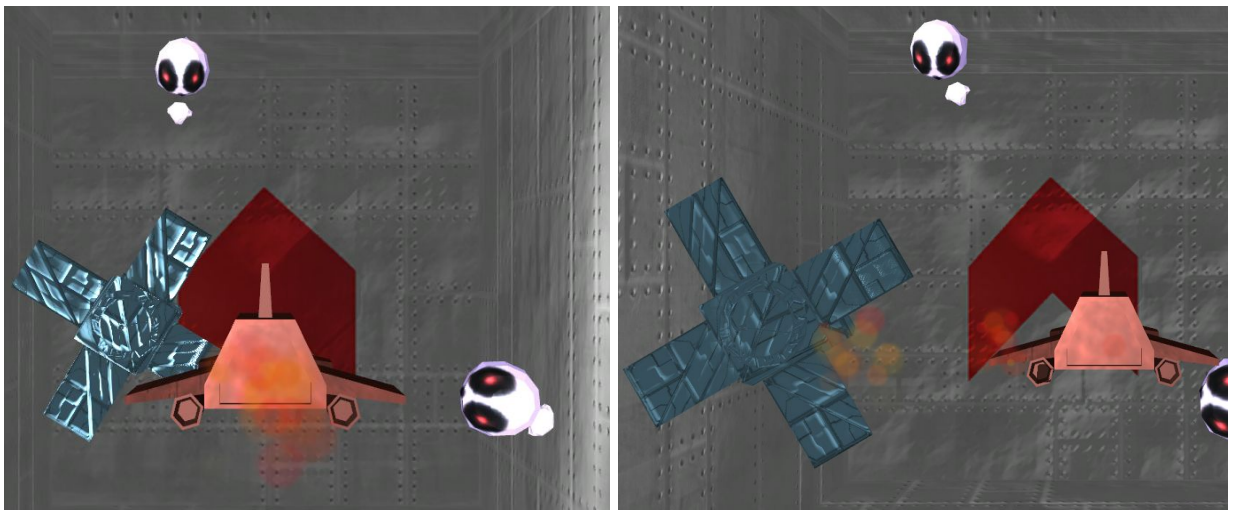


- **Shadow Mapping + PCF**: The main pipeline of ShadowMapping can be found in ShadowMapManager.cpp. First, the depth values of the ship, the ghosts and the plane obstacles are rendered into a framebuffer. The texture that was attached to the depth framebuffer is then used for shadow mapping during the normal rendering phase of the involved objects. For our shadow mapping, we decided to use a spot light which covers all the objects of the level. To see the depth buffer, hit 'Z' on the keyboard as described later in the chapter 'Controls'. The tutorial sources used to implement this effect are:
  - CGUE slides
  - OpenGL Superbible Sixth Edition
  - http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping
  - http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/
  - http://ogldev.atspace.co.uk/www/tutorial23/tutorial23.html
  - http://ogldev.atspace.co.uk/www/tutorial24/tutorial24.html
  - http://ogldev.atspace.co.uk/www/tutorial42/tutorial42.html

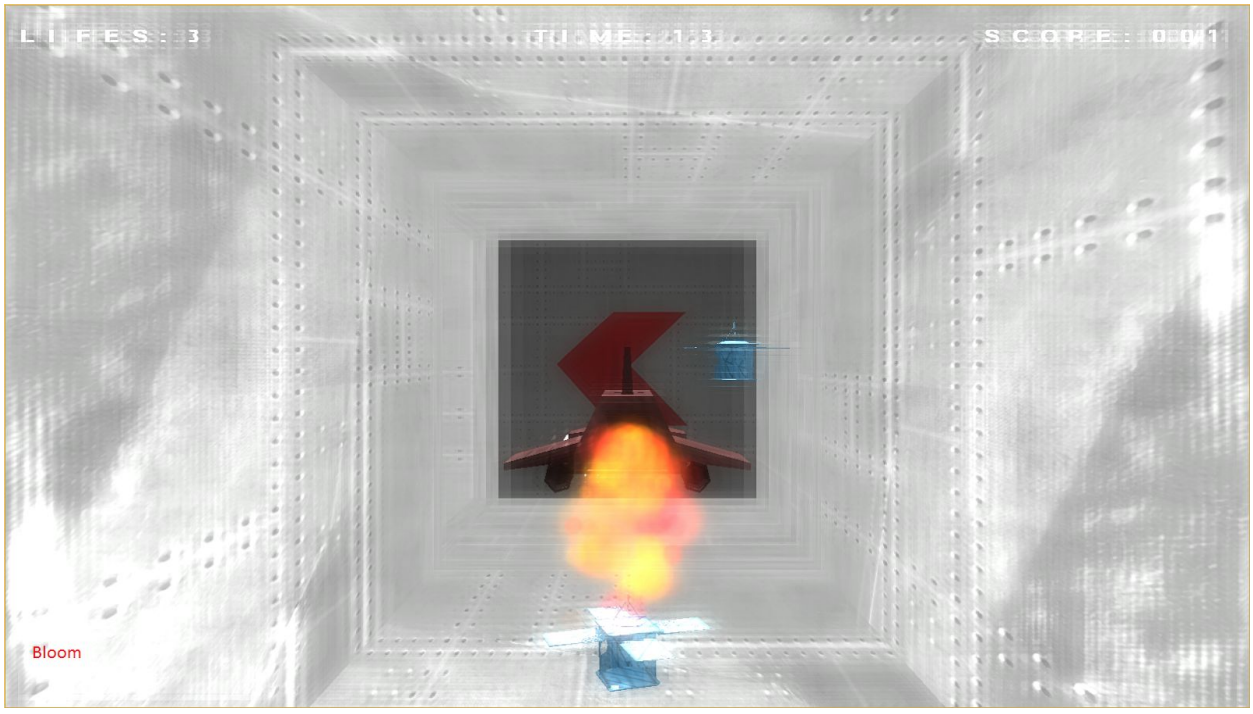


Shadow Mapping

- **Normal Mapping**: The normal mapping effect is implemented for every satellite (in this case combined with the shadow mapping effect) and can also be seen at the borders of the tunnel system. The tangent, bitangent calculation and VBO indexing are implemented in SceneObject.cpp functions calcTangents(), computeTangentBasis() and vboIndexer.cpp function indexVBO_TBN(). The data needed to perform these calculation are provided by the object loader of the scene objects. The tutorial sources used for normal mapping are:
    - http://learnopengl.com/#!Advanced-Lighting/Normal-Mapping
    - http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/



- **Bloom:** BloomManager.cpp includes the the main implementation of Bloom, while Game.cpp only calls the various functions of BloomManager. In the first step, we render the scene into framebuffer in order to get a texture, which is then filtered with a bright-pass shader to get all bright parts of the scene. Then, the new texture is blurred vertically and horizontally and then blended additive with the original texture. The resulting bloomed scene is rendered to the screen with the help of a full-screen quad. Bloom is usually activated, when a ghost obstacle collides with the ship, but can also be activated manually with the keyboard, pressing ‘B’, as described later in ‘Controls’. For the implementation, following tutorial sources were used:
    - CGUE slides
    - http://prideout.net/archive/bloom/index.php
    - http://http.developer.nvidia.com/GPUGems/gpugems_ch21.html

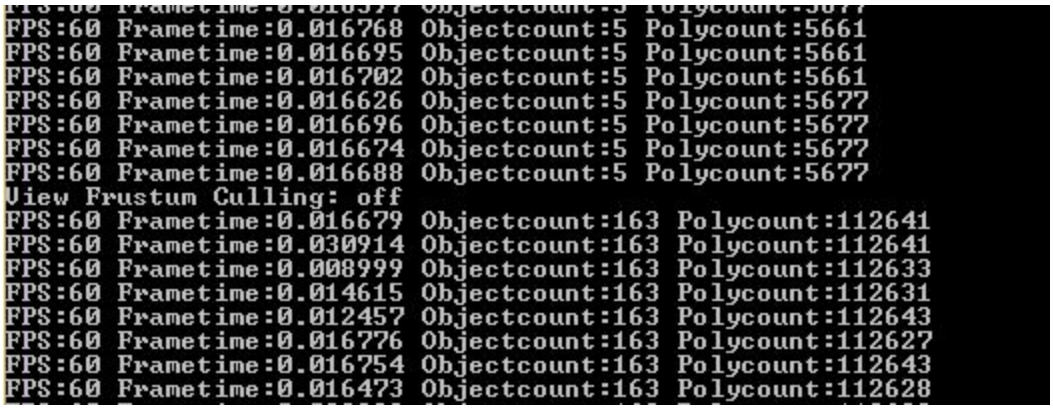


Animated Objects

The animated objects in our game are the ghosts. While the ghost moves up and down, the hand of the ghost does so too. In addition to that, the hand rotates around the ghosts’ body. The rotation direction of the hand (left or right) is determined by a randomized variable.

Frustum Culling

You can recognize, that view frustum culling is implemented, when hitting F2. So you see in the console how many objects are currently rendered. We have a class AABB.cpp for the Axis-Aligned Bounding Box and one class Frustum.cpp for the view-frustum. the functionality itself is implemented for obstacles and satellites, since these two objects are the only ones, that not allways are shown.



Controls

You control your spaceship with the w,a,s,d and arrow keys of your PC-keyboard. You steer the spaceship upwards, to the left, downwards and to the right using the w,a,s,d keys and change directions upwards, left, downwards and right with the arrow keys. The spaceship has a certain velocity, and is moving forwards automatically. Satellites are hacked by bypassing.

Key	**Effect**
W,A,S,D	Steer up, left, down, right
Arrow Up, Left, Down, Right	Change directions upwards, left, downwards, right
M	mute or unmute the sound
ESC	Quit game
F1	pause game
F2	show FPS ingame and additional informations in the console. (e.g. frametime and the number of rendered objects)

F3	turn wireframe on or off
F4	change the texture sampling quality between nearest neighbor and bilinear
F5	change the mip mapping quality between off, nearest neighbor and linear
F8	turn viewfrustum culling on and off
F9	turn transparency on and off
B	turn Bloom on and off
Z	turn depth buffer rendering on a small quad on and off

Implementation

- The camera is not freely movable, because it should only follow the spaceship's trail. The position is dependent to the position of the ship.
- The only moving object is the spaceship, which moves automatically towards the direction it faces and can change position and direction according to the user's input (see "Controls").
- The game is capable of reading obj, texture and bitmap files, so object normal vectors are provided and every object in the scene has a texture. There is a global light source for the whole scene.
- Bullet physics is used to detect collisions between the ship and obstacles. Detect a crash into the wormhole or recognize when a satellite is hacked.
- Some of the models are from different sources of the internet. e.g. http://www.3dvia.com/models/04102C3A0C1E3002/simple-low-poly-cargo-spaceship. All models are reworked or created by ourself in Blender.

Features

- Intuitive controls
- Start, Loading, Game Win and Game Over Screens
- Object and image loading
- Collision detection with Bullet physics

Libraries

Following Libraries were used and included in the project:

- **GLEW:** http://glew.sourceforge.net/
- **GLFW**: http://www.glfw.org/
- **glm**: http://glm.g-truc.net/0.9.6/index.html
- **Bullet physics:** http://bulletphysics.org/wordpress/
- **config4cpp**: http://www.config4star.org/
- **FMODEx:** http://www.fmod.org/download/#Previous