

A Birds Eye View Documentation

The program was tested on a lab PC using an NVIDIA GTX 1060 graphics card.

Controls:

Cursor keys (or WSDA keys) move the camera. Camera direction is controlled with the mouse.

Up/Down cursor keys: move camera forward and backward

Left/Right cursor keys: strafe camera left/right

Q/E keys, mouse x-axis: turn camera left/right

mouse y-axis: turn camera up/down

ESC: exit program

F: switch between fullscreen and windowed mode

F3: toggle wireframe rendering mode

F6: toggle movement of light (enabled by default)

F7: draw shadow map only (for debugging, disabled by default)

C: switches between the shadow maps that are drawn in debug mode

F8: toggle view frustum culling (enabled by default)

F10: toggle shadow mapping (enabled by default)

Keypad * and /: double/half the squared screen space error threshold (default and minimum is 1 pixel)

1, 2, 3 keys: toggle display of terrain in shadow map cascade 0, 1, 2, respectively. If view frustum culling is disabled, only the rendering of the respective shadow map is disabled.

P: start/stop flight path recording for automatic camera, will overwrite "cameraPath.bin" when stopped

M: start/stop playback of automatic camera, only works while not recording

Shadow Mapping:

Inspired by <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/>

Depth Texture Creation:

First the scene is rendered into a depth texture from the view direction of the directional light illuminating the scene.

This is done by setting up an orthogonal viewbox for parallel light projection that is rotated towards the player position in direction of the directional light.

Then the whole scene is rendered with this new MVP matrix, while the depth is written into a texture inside of the fragment shader.

Camera Projection Shadow Rendering:

This texture then is passed to the shader which renders the whole scene, where the depth of a pixel in camera projection space, which was transformed into light projection space, is compared the depth of the corresponding fragment inside the depth texture.

If the fragment in projection space is behind the fragment in the depth texture, then the fragment is in the shadow of an object in the scene, therefore the visibility of the fragment is reduced, producing the shadows.

Cascaded Shadow Mapping:

Inspired by <http://ogldev.atspace.co.uk/www/tutorial49/tutorial49.html>

Cascaded Shadow Mapping extends the concept of shadow mapping for the use of multiple shadow maps.

The idea behind it is that shadows that are nearer to the camera required more details and are more visible and therefore required lower resolution, while the edges of far away shadows would not be visible if the shadows resolution is higher than the spatial resolution of the shadows on the screen.

Furthermore we could only render those objects into the shadow map, whose shadows are visible in camera space.

Therefore we split the view frustum of the camera into multiple sections for the shadow cascades and calculate the projection and view matrix for each cascade.

This can be done by first transforming the boundaries of the frustum sections from camera view space to world space coordinates by utilizing the inverse camera view matrix.

Then the boundaries can be projected into the lights view space by utilizing the lights view matrix based on its direction.

Now these boundaries in light space can be used to construct the depth-view-projection matrix for each section, which uses an orthogonal box as a projection matrix.

The next change for Cascaded shadow mapping is taking place in the fragment shader, where the z-value of the vertex in camera space compared to the camera space z-values of the segment ranges to select the appropriate cascade.

Further Poisson filtering is applied to smooth the shadow edges.

By applying Cascaded Shadow Mapping the shadows appear in detail while optimizing, by only rendering regions that are visible.

Skybox:

Inspired by <https://learnopengl.com/Advanced-OpenGL/Cubemaps>

A inward facing cube model is generated and texturized by the 6 faces of a skybox. It is then shaded before rendering the whole scene to provide the appearance of a rich environment in the distance.

Phong Shading with directional light:

Inspired by <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-8-basic-shading/>

Render texture to screen quad:

Inspired by <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/>

Terrain Tessellation and Heightmap Geometry:

Tutorial used: <http://ogldiv.atspace.co.uk/www/tutorial30/tutorial30.html>

The Terrain is represented as a height map and partitioned into a set of tiles. It is first simplified across several levels of resolution and for each tile and level the object space errors between the original and reduced resolution are pre-computed. The simplified meshes are discarded and the object space errors are used to determine the level of detail for the terrain as follows.

Each tile has 8 x 8 blocks, and for each block and resolution level the maximum object space error is projected to the screen to determine the screen space error. Then each tile is partitioned into a quadtree such that each node covers one or more blocks and determines the resolution of the blocks.

The nodes of the quadtrees are sent to the tessellation shader as patches and split into triangle meshes according to their tessellation levels. Each vertex of the tessellation result is finally displaced by a lookup in the height map.

Quadtree construction and Tessellation:

Quad tree construction is done for each tile separately and in two steps: first the object space errors for each block are projected to screen space to determine the lowest LOD level at which the screen space error is below the threshold. From this 8x8 matrix of LOD values a 4x4, 2x2, and 1x1 matrix is derived that contains the maximum of a 2x2 subset of the previous matrix, similar to an image pyramid. This pyramid is then used to create a quadtree. If the LOD of the 1x1 matrix is low enough this is used as the LOD of a single node for a tile. Otherwise the node has to be split (given the fact that the maximum tessellation factor supported by hardware is smaller than the maximum resolution of a tile) and each child node

is checked against the 2x2 matrix. This process is repeated if necessary to the 8x8 block level.

The inner tessellation factor for each node is simply determined by its LOD value. The outer tessellation levels for each node need to take into account its own LOD and all neighbouring nodes (also from other tiles if the node lies on a tile border). To avoid the time-consuming search for neighbours the calculations traverse the nodes' edges instead of the nodes themselves. This starts with the edges between two neighbouring tiles (i.e. their top-level nodes). If both are leaf nodes, the factor can be calculated directly. If only one is a leaf node, its neighbours can be found easily by traversing the children of the other node adjacent to the edge, and the factor is calculated as described in the paper. If both nodes are inner nodes, the tessellation factor is calculated independently for their two children adjacent to the edge. During the traversal of child nodes the outer tessellation factor for their shared edge is also computed along the way, ensuring that each edge is processed exactly once.

Preprocessing of terrain DEM data:

The terrain DEM (digital elevation model) data is taken from <https://data.opendataportal.at/dataset/dtm-austria> (DTM Austria, 50 m, only included in the source package due to size limitations). The preprocessing is done when the program starts and there is no preprocessed data yet, and includes:

- computation of vertex normals,
- object-space errors for each LOD and block,
- minimum/maximum height values for each tile and sub-tiles (required for view frustum culling).

The computed data is stored in compressed form (using half-floats) for reuse in a subsequent program execution, in the “textures” directory in files with the “.dat” extension (only included in the binary package).

View frustum culling for terrain:

View frustum culling for the terrain is done in two steps: first each tile is checked against the view frustum, resulting in a set of “active” (i.e. visible) tiles. For each active tile a quadtree is constructed and the patches in the quadtree are culled as well.

The method used to implement culling is described in “Real-Time Rendering, Fourth Edition”, chapter 22.10.1

View frustum culling is done for each cascade of the shadow map and for the camera view using a cascaded view frustum (an extended view frustum with multiple near and far planes). This way only the part of the terrain that lies in a shadow map cascade is rendered to the respective shadow map.

Culling against the view frustums of the shadow map may include parts of the terrain that are outside of the camera view. This is because a patch of terrain that is off-screen might cast a shadow on the scene, which would be lost if the terrain was culled only to the main camera frustum during shadow map rendering.

Additional Libraries used:

The libraries for linking can be found in the source folder External/libs/x64/Release, and the respective DLLs are in the folder DLLs/x64/Release.

GLFW library:

<http://www.glfw.org/>

The GLFW library has been used creating windows, contexts and surfaces and for receiving input and events.

GLEW (OpenGL Extension Wrangler) library:

<http://glew.sourceforge.net/basic.html>

The GLEW library has been used in glewExperimental mode to create a valid OpenGL rendering context.

GLM (the OpenGL math library)

<https://glm.g-truc.net/0.9.9/index.html>

GLM is used throughout the program, mostly for matrix and vector calculations.

Mango Framework (image library part)

<https://github.com/t0rakka/mango>

The Mango image library is used for loading the textures for the game objects. It replaces the ImageMagick++ library previously used.

Assimp (Asset import) library

<http://assimp.sourceforge.net/>

To load .obj and .mtl file into the program the assimp library was used. The objects were then rendered by us.

TIFF library

<http://www.libtiff.org/libtiff.html>

The original height map of the terrain is loaded with libtiff compiled with support for floating point data types.

Visual C++ Redistributable for Visual Studio 2015

<https://www.microsoft.com/en-us/download/details.aspx?id=48145>

This includes the DLLs msvcp140.dll, vcruntime140.dll and vcruntime140_1.dll, which are required at runtime since the program was developed with Visual Studio 2019.

Textures and Models:

Hummingbird 3D model: <https://free3d.com/3d-model/hummingbird-84681.html>

Terrain texture: <https://de.brusheezy.com/texturen/20185-nahtlose-grune-gras-texturen>

Skybox texture: <https://opengameart.org/content/sky-box-sunny-day>