1 Overview

This report presents our advancement on the third part of the project : interaction and animation. During all the run of the project, we focused on code architecture and quality rather than quantity. It would have been impossible to implement every single idea we had about the project anyway.

We did implement all the basic

2 Implementation

2.1 Basic - Fly through mode

In order to freely fly through our scene, the camera mode free fly provide 3 possible actions:

- go forward/backward.
- turn left/right.
- turn up/down.

To have all possible rotation, the camera has to rotate on virtual axis, see figure 1 present the behavior of a user turning up and then right creating a virtual axis of rotation. To

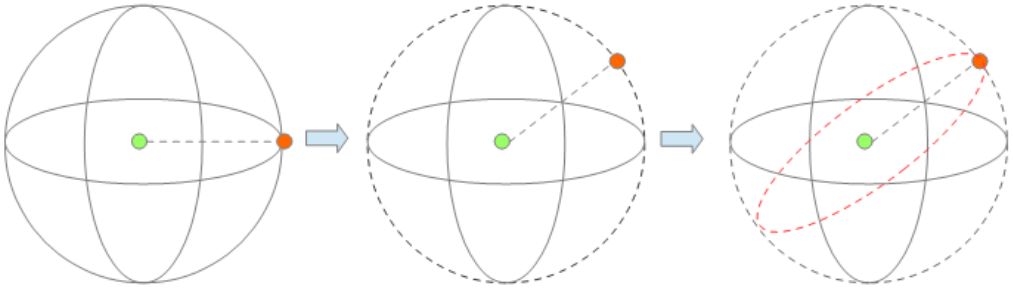


Figure 1: Free flying rotation example

achieve such rotations,the look at point has to be displaced staying always at equal distance to the camera position. We proceed this rotation in 3 steps: translate to origin, undo previous rotations, do new rotations, return to initial placement. Rotations are proceeded only in two axis, Y and Z, using rotation matrix computed according to inputs angle. A velocity has been added, the acceleration is gradual until a maximum value, same behavior for deceleration.

2.2 Basic - FPS exploration mode

The FPS exploration mode has slightly different rotation behavior. Camera's rotations are expected to reproduce similar effect as the head rotation of a human body. In this purpose the lateral rotations will always be proceeded on a virtual axis parallel to the world horizontal axis. An example of such behavior is represented figure 2. Thus, transformation is simplified for lateral rotations compared to free flying mode. Vertical rotation stays similar. In order to have a smooth walk in our scene, the Z component of the camera

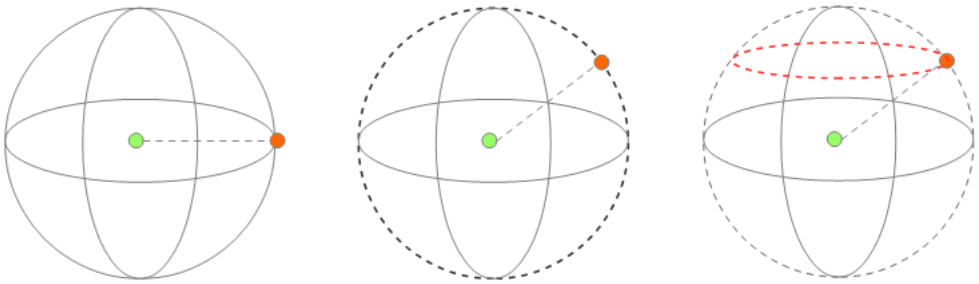


Figure 2: Fps rotation example

position is equal to the mean of the 24 positions around camera x-y position, avoiding camera shaking in sharp terrain.

2.3 Advanced - Physically realistic movements

In order to have movement physically more realistic, an acceleration/deceleration has been added to every movement, body will not instantaneously move. An additional running function has been added to increase a maximum forward speed higher. A simple jump function has been created, it uses a sinus to shape the jump z variations. Jumping speed vary according to the jump stage, beginning/ending of the jump has a higher speed then higher steps.In the case of jumping from the top of a mountain, the falling speed will reach a maximum value and keep falling at constant speed until it reach the floor. While jumping the trajectory is computed using the action settled just before the jump. For example, if a user was turning, the rotation will continue during jump making a spinning effect, which will decelerate once the users reach the floor again.

2.4 Advanced - Camera path control

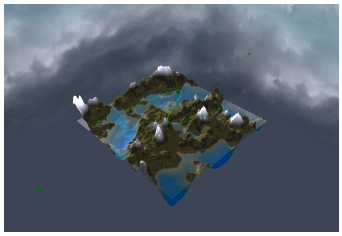
The camera trajectory has been modeled using cubics Bezier curves, therefor we implemented several computation methods:

- Bezier with 4 control points using de Casteljau algorithm (see figure 3a)
- subdivision algorithm with 5 splits at 0.5 (very similar result as 4 point de casteljau, see figure 3b)
- concatenated Bezier curves with infinite maximum of Bezier curves (see figure 3c and 3d)

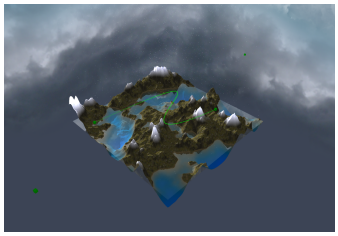
In order to observe correctly the Bezier curves, we render every control points and the path of each curves. The curves can be edited during execution, to know the controls to set it please refer to section "how to use".

2.5 Advanced - Pictorial camera

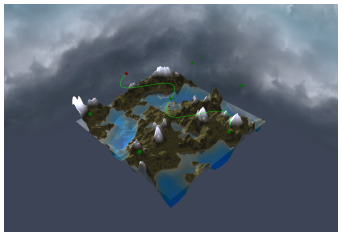
On top of the bezier curve, we added a pictorial representation of the camera. This camera allow the user to previously see what will be the camera path and rotations while moving along bezier curves. The pictorial camera is shown figure 4



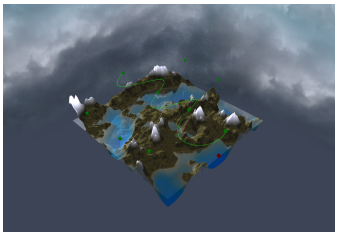
(a) 4 points Bezier



(b) Bezier using subdivision algorithm



(c) Two bezier curves concatenated



(d) Three bezier curves concatenated

Figure 3: Bezier curves

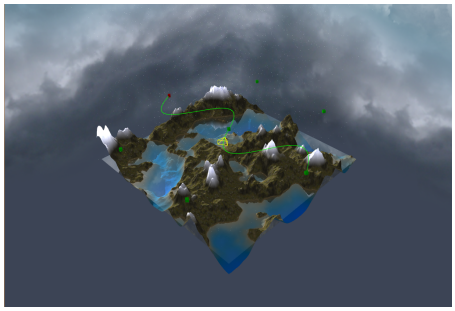


Figure 4: Pictorial camera example

2.6 How to use

Along the project, many different controls has been created to use several functionality, we will present them in the following section. The user can change exploration modes using numerical pad:

- Key "1": free flying exploration mode.
- Key "2": first person exploration mode.
- Key "3": camera follow bezier curves.
- Key "4": trackball mode.

2.6.1 Free flying controls

During free flying, user can use followings controls:

- Key "W": go forward.

- Key "S": go backward.
- Key "A": turn left.
- Key "D": turn right.
- Key "Q": turn up.
- Key "E": turn down.

If during free flying user press key "ENTER", it will drop a control point to create a bezier curve, then the controls will be slightly modified: key "W" will increase the distance between camera and control point, key "S" will decrease this distance and rotation controls will stay similar. Pressing ENTER again will then set the control point. The user has now settled 2 control points, when he will change position, a bezier curve preview will be rendered. At the new location user can press ENTER again to set the 2 last control points and repeat this operation as much as he wants.

2.6.2 FPS exploration controls

The rotation control stay similar as free flying exploration with additional controls: key "X" will start a jump, if at this moment the user was moving, this move will influence the jump. During jump, user can't change rotation or speed until he reach the floor again. Key "LEFT SHIFT" while going forward will increase its speed (running).

2.6.3 Bezier curve control

User can freely modify and create new control points with the followings controls:

- Key "Z" : Select next control point to modify.
- Key "U" : add 0.1 to position X.
- Key "J" : remove 0.1 to position X.
- Key "I" : add 0.1 to position Y.
- Key "K" : remove 0.1 to position Y.
- Key "O" : add 0.1 to position Z.
- Key "L" : remove 0.1 to position Z.
- Key "+" : add a new bezier curve at the end of the existing.
- Key "-" : remove bezier curve at the end of the existing.
- Key "7" : Start/stop pictorial camera animation.

During user modifications, curve will adapt its shape to respect a continuity constraint.

2.7 Advanced - Water modeling

Water refraction, reflection and depth

For the water reflection effect, we should render the terrain from a flipped camera position. The terrain is thus rendered two times. It is rendered a first time with the real camera position to the default framebuffer (screen) and a second time with a flipped virtual camera position to a texture which is attached to a framebuffer object. The texture is later used to render the water surface.

2.8 Advanced - Particle system

The cheapest way to display particles is to use sprites: polygons that face the camera in any orientation.

As for the heightmap, if we were able to work with OpenGL 4, we could have use compute shaders instead of an empty vertex shader for this kind of GPGPU computation. It would have permit us to store particle data in buffer objects and avoid pingpong storage.

transparency gradient proportional to distance to center : `gl_PointCoord` particle size inversely proportional to the squared distance to the camera : `gl_pointSize` no sorting with respect to camera distance for transparency : can create weird effect if a particle that is nearer than another gets drawn before. The farther particle will visually seem to be in front of the nearer. Thanks to particle speed, this won't be visible. Draw a real flocon ? not much added value to particle speed Use geometry shader ? Particle texture ? A common technique to solve this is to test if the currently-drawn fragment is near the Z-Buffer. If so, the fragment is faded out. However, you'll have to sample the Z-Buffer, which is not possible with the "normal" Z-Buffer. You need to render your scene in a render target. Alternatively, you can copy the Z-Buffer from one framebuffer to another with glBlitFramebuffer.

As we cannot read from a texture and render to it at the same time, we need two velocity textures. The one used as an input is the one that was rendered to before. At each frame rendering we exchange the two.

The vertex attribute buffer contain integers from 0 to nParticles - 1. The attribute is the index in the `particlesPosTex` which will allow the `particles_render_vshader` to retrieve the particle position. The attribute type is `GL_INT`. As for the indices to not be converted to floating point values, the `glVertexAttribIPointer` function is used (notice the `I`).

The `gl_VertexID` vertex language input variable is used by the `particule_render_vshader` to index the particle textures. 1D texture to store position and velocity. Window and textures are of size `nParticles` times 1 pixels.

In our implementation, particles that fall below the water level are reseted to the top and will fall down again, indefinitely. In order to keep the aggregate compact and so as it does not spread out in the space, particles that go too far in x or y direction get moved to the oder side.

2.9 Advanced - Two simultaneous views

By two simultaneous views, we mean the rendering of the seen from two points of views. In our case, a control view and a camera view (fig 5). The control view is used to set the camera path Bézier curve control points and have a global view with an animated pictorial

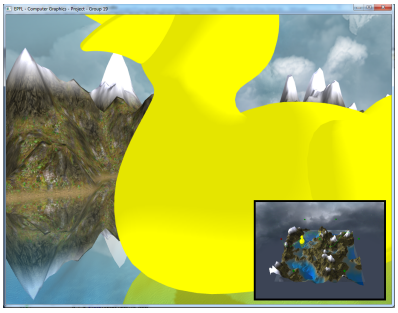


Figure 5: Illustration of our two-view system.

camera. The camera view actually shows what the camera sees. The keyboad ENTER key can be used to exchange the two views

From an implementation point of view, almost each object of the scene (skybox, terrain, water, shadowmap, skybox, particles) should be drawn from two different points of view. They are thus rendered to two FBOs (with attached output texture and depth buffer) instead of the default framebuffer (the screen). The `Display` class eventually uses the texture to assemble the final screen view.

For performance reason, it is crucial to render the two points of view one after the other as shader program switch is expensive.

- The linear filtering is only used for the small preview.
- As it creates too much aliasing, the main image is direclty accessed with texelFetch which does bypass the filtering.

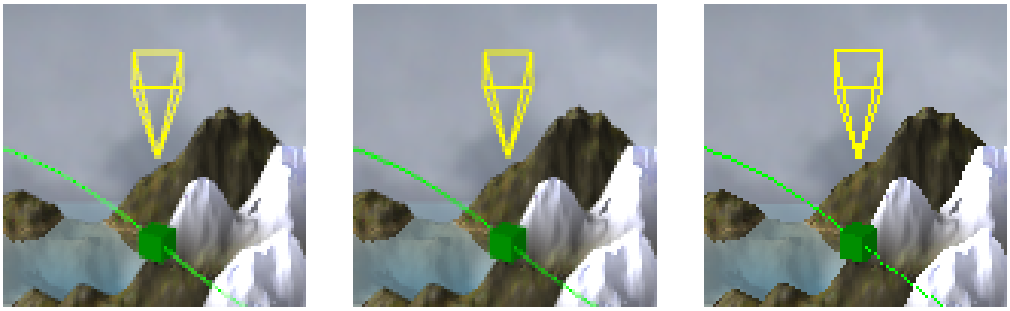
To produce hight quality rendering, the two views were rendered to multisampled textures (`GL_TEXTURE_2D_MULTISAMPLE`), using four samples per pixel. Multisampling, also known as multisample antialiasing (MSAA), is one method for achieving full-screen antialiasing (FSAA). This technique was introduced by `ARB_texture_multisample` In order to improve performance, multi-sampling was disabled for the default framebuffer in the `glfwCreateWindow` helper function. It makes no sense to do the multi-sampling twice.

The multi-sampled texture is manually resolved in the post-processing shader.

Depth is renderbuffer as we won't touch it again.

Renderbuffer Objects are OpenGL Objects that contain images. They are created and used specifically with Framebuffer Objects. They are optimized for use as render targets, while Textures may not be, and are the logical choice when you do not need to sample (i.e. in a post-pass shader) from the produced image. If you need to resample (such as when reading depth back in a second shader pass), use Textures instead. Renderbuffer objects also natively accommodate Multisampling (MSAA).[1]

[1] https://www.opengl.org/wiki/Renderbuffer_Object



(a) Automatic resolve (b) Manual resolve (c) No MSAA

Figure 6: Multi-sampling anti-aliasing

3 Improvements on last stage

3.1 Code cleanup

As this hand-in is the last opportunity to modify our code, we put great effort in improving its quality by cleaning it up, including comments. A great job was done to functionalize shader code. As a side effect, we did also discover some little mistakes. We did also refactor most of the code to encompass our new object oriented architecture introduced in stage 2. In overall, code quality has greatly improved.

Redeclare the GLSL built-in blocks : `gl_PerVertex` to be able to use separable program objects.[2]

Use `glProgramUniform` instead of `glUniform`. The former takes a program ID argument which indicates in which program it should retrieve the uniform ID. Using this function it is no more needed to have the program marked as used by `glUseProgram`.

By testing our code on Window and Linux with a combination of ATI and Nvidia hardware and drivers, we found quite some interesting corner-cases where the OpenGL specification is not entirely met. An example is the size of the `gl_ClipDistance` array of the `gl_PerVertex` block. According to the spec[3], the

3.2 Vertices object

To achieve better modulation, we have separated the vertices creation code into a class hierarchy. The `Vertices` base class is an abstract class, declaring only virtual methods, which defines the interface. The `VerticesQuad`, `VerticesGrid` and `VerticesSkybox` inherit from it and implement the `generate draw` and `clean` methods which are specific to them. This design allows more than one RenderingContext object to share a `Vertices` object. The `Terrain` and `Shadowmap` makes use of this. It also offers better source code modularity.

[2] http://www.geeks3d.com/20130106/nvidia-updates-its-opengl-sdk/
[3] https://www.opengl.org/sdk/docs/man4/html/gl_ClipDistance.xhtml

3.3 Lightning

Phong shading was improperly implemented at stage 2. The material texture was only used to compute the ambient color. The diffuse and specular colors were fixed for all the terrain. Probably a reminiscent of stage 1. This is now fixed. We first retrieve material color properties from textures. The retrieved color is then split across the three lightnings (ambient, diffuse and specular) with the help of coefficients which some up to 1. The specular lightning coefficient is non-zero for water only. The water uses two normals : a normalmap for diffuse lightning (this is used to create the impression of water movement) and a (0,0,1) normal for specular lightning (this is used to create the impression of sun light reflection).

The light direction was defined in stage 2 to be the same for all the vertices. This defines it as a directional light. It was changed to a spot light (or point light in the lightning context), to be coherent with shadowing. The shadowmap is indeed computed using perspective projection, not orthographic.

3.4 Shadowmap

We have improved a lot the shadowmap, as it was not yet working properly at Stage 2. Figure 7 shows how it works by showing the shadowmap, the distance to light and the rendered terrain for three different light positions. While we were at shadowmap, we also took the time to test some parameters.

The division by the w component to apply projection should be done before the coordinates change from (-1,-1)x(1,1) to (0,0)x(1,1). It can thus not be integrated in the `lightMVP` matrix and should be computed by the shader. To save computations and GPU memory bandwidth (passing a `vec3` instead of a `vec4`), this can also be computed by the vertex shader. The result is then interpolated and passed to the fragment shader. The differences are very small, as shown by figures 8a and 8b.

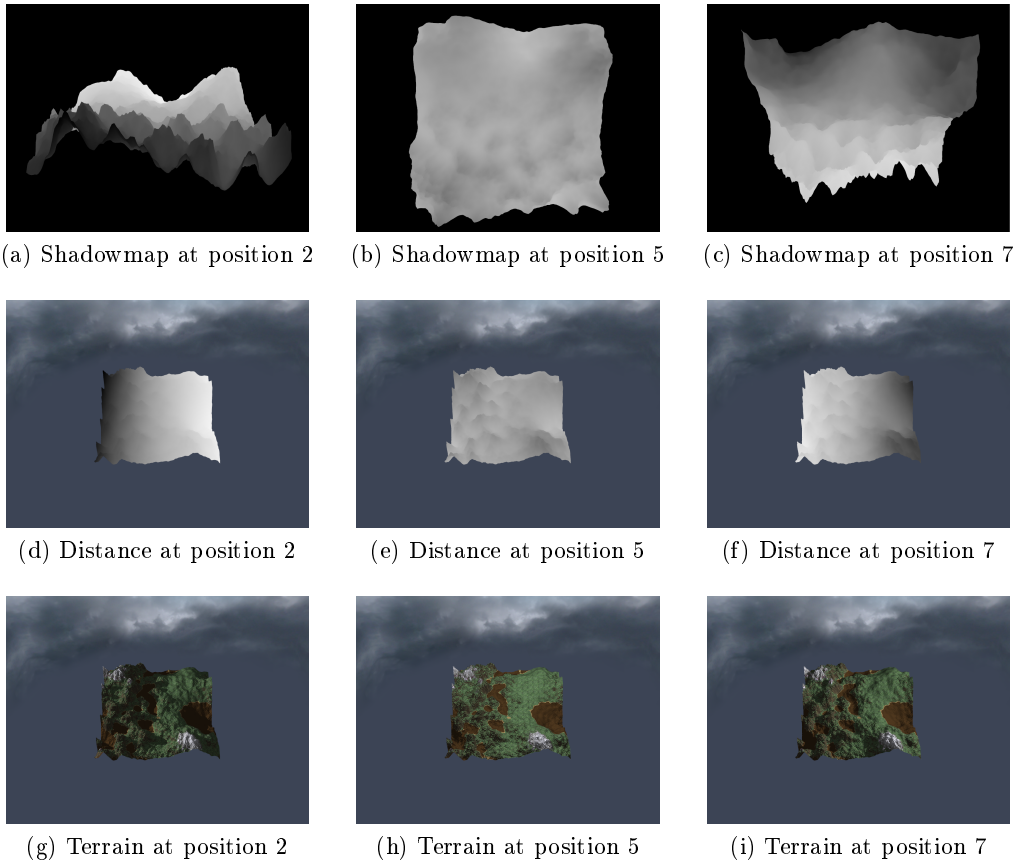
The choice of the bias, used when comparing the distance to light with the one stored in the shadowmap has an impact. A too low value will result in Z-buffer fighting (fig. 9a) while a too big value will result in shadow errors (fig. 9c). A good compromise should be found (fig. 9b).

Percentage closer filtering (PCF), a method which sample four neighboring pixels and return an average of the tests, greatly smooth the transition between shadow and light regions. Figures 10b and 10a show the same scene with and without PCF. The effect on transitions is clear.

The differences between a 32 bits depth buffer (fig. 10b) or a 16 bits (fig. 10c) depth buffer is very little. We have thus chosen 16 bits to save computation time and memory space.

3.5 Heightmap

The heightmap texture was configured with the default heuristic for out-of-range values. This default is to wrap around the texture (`GL_REPEAT`) which creates artifacts at the borders of the terrain due to linear filtering which access neighboring pixels (fig. 11a). Clamping the texture coordinates (`GL_CLAMP_TO_EDGE`) to the [0,1] range eliminates the artifacts (fig. 11b).



(a) Shadowmap at position 2 (b) Shadowmap at position 5 (c) Shadowmap at position 7

(d) Distance at position 2 (e) Distance at position 5 (f) Distance at position 7

(g) Terrain at position 2 (h) Terrain at position 5 (i) Terrain at position 7

Figure 7: Shadowmap construction

The heightmap code was also refactored to accommodate our new object oriented architecture. This resulted in a cleaner design and better readability as well as some improvements of the generic code.

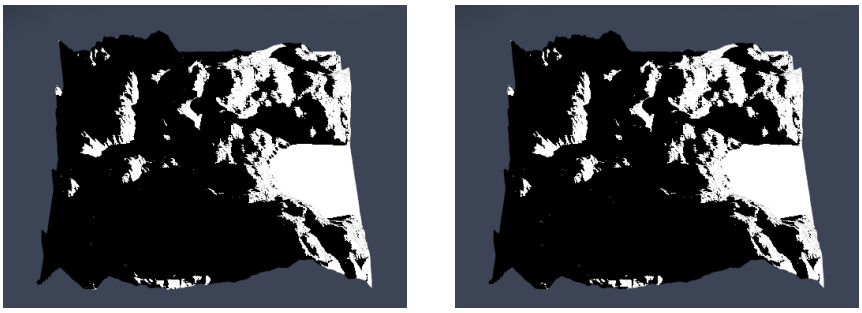
Another improvement of the heightmap code is that the `position` vector is now a `vec2` instead of a `vec3` as the vertex z position of a quad is 0 anyway. This saves some GPU memory bandwidth.

3.6 External Objection addition

We tried to render together an external object (a yellow duck) by loading its 3D mesh. We did some simple transformation to place it in right position and move linearly.

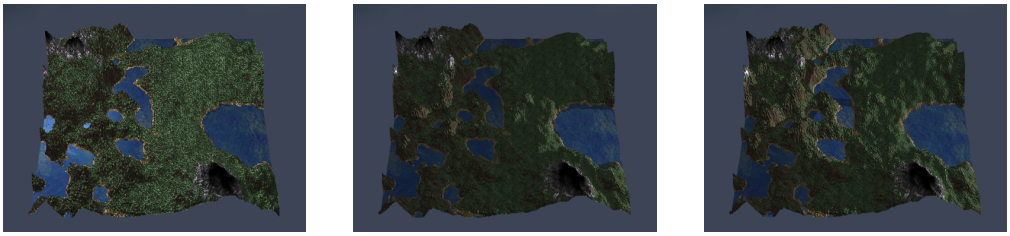
The reflection of the duck is also rendered.

4 Results



(a) In fragment shader (b) In vertex shader

Figure 8: Coordinates transform

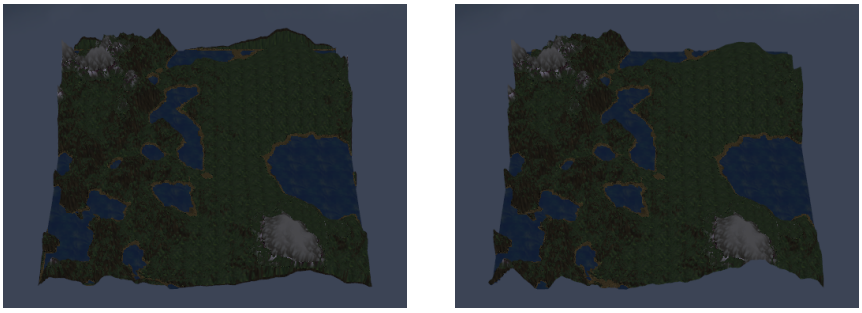


(a) No bias (b) Good value (c) Too big value

Figure 9: Shadowmap with different bias values

(a) 32 bits depth without PCF (b) 32 bits depth with PCF (c) 16 bits depth with PCF

Figure 10: Shadowmap



(a) With artifacts at the border (b) Without artifacts

Figure 11: Heightmap