Karlsruhe Institute of Technology
Computer Graphics Group

Graphics Programming Lab
Assignment 4

Terrain Rendering

Abstract

This assignment introduces a simple yet efficient terrain rendering method, which allows rendering a large terrain using an efficient multi-resolution level-of-detail technique entirely on the GPU.

1 Introduction

There are plenty of terrain out-of-core rendering techniques nowadays, we will not discuss all of them. The interested reader can refer to the survey [1] for the overview.

In this assignment we will focus on a modern and popular rendering technique based on dynamic tessellation on the GPU. The terrain rendering should be done on top of your **existing** personal Qt framework developed by you during the previous assignments.

1.1 Terrain Rendering Basics

By the *terrain rendering* we usually mean the visualization of an surface represented by a *height map*[1]. The height map is a 2-dimensional horizontal regular grid with the surface elevation stored in every cell. In graphics, the height maps are usually represented as single-channel 2D textures[2]. The height map defines the surface of the underlying terrain.

The height map can be:

- Procedurally generated (e.g. with the noise or fractals);
- Manually modelled in specialized authoring or sculpting tools;
- Captured from the real geodesic data.

These approaches can also be combined with each other. For example, the real geodesic data can be used for the rough surface representation, accomplished with some procedurally generated noise for achieving local detailed roughness of the surface when the camera approaches the terrain.

[1] In this case the terrain cannot have multiple layer, like caves, tunnels etc. Rendering these is another interesting problem, which is solved by other techniques or extensions.

[2] The 8-bits precision of the common color channel is usually not enough for the precise representation of the elevation

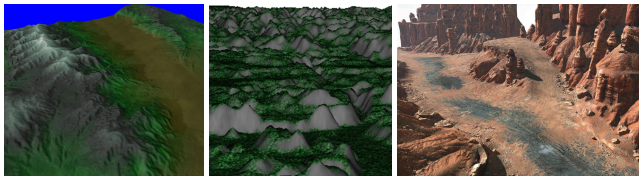

Figure 1: *An example of different types of terrain materials placement techniques (left to right): height-based procedural material placement; slope-based procedural material placement; authored mask texture. Please zoom into the electronic version for details.*

1.2 Terrain materials

Besides the elevation itself, the height map can be also enriched with information about different materials (e.g. rocks, sand, grass etc.). Storing all such details of the terrain surface using a single huge texture would require a lot of memory, an intensive I/O and would lead to a redundant level-of-detail at the distance.

Instead, for describing the structure of every material, the *texture splatting* is usually employed. The texture splatting is the way multiple detailed facture textures are combined (usually modulated) by means of the weights (see Figure 2). Each facture texture is a tiled texture describing some local properties, or *facture* of a particular material. This technique allows to achieve the rich visual look while using only moderate amount of memory.

The decision on mixing different materials can be made using different techniques:

- The coarse *weighting texture* is stored in the same way as the elevation height map; it contains mixing weights for every material
- Mixing weights are procedurally created based on different geometric features of the terrain (e.g. height-based mixing, slope-based mixing)

For the illustration of the different approaches for material mixing see Figure 1.

1.3 Rendering terrain with tessellation shaders

In large open-space scenes, the geometry of terrain can require a significant memory consumption and a huge rendering effort. Thus an efficient representation of the terrain geometry is a crucial part of terrain rendering technique. We will use hardware-accelerated

dynamic tessellation for this. Tessellation units have been available in GPUs for a few years now.

We start out with the basic idea. Imagine the camera standing on the terrain. Despite of the fact that the terrain surface has different levels of elevation, we can still treat it as a relatively flat surface in a large scale, as most of features (like mountains, hills, cliffs etc.) are local. Thus the closer the part of the terrain to the camera, the more detailed it should be. If we split up the visible terrain surface according to the distance to the camera into several *zones*, we could assign a dedicated level of details to each zone. For example, we can split it up into three zones: zone 1: 0 – 50 meters, zone 2: 50 – 500 meters, zone 3: 500 – 5000 meters. Then we can make each farther zone's triangle mesh coarser than for the preceding closer one. For the convenience of joining them, we make the tessellation of every farther zone twice coarser than the tessellation of its preceding zone. Modern GPU hardware allows us to tessellate geometry adaptively during rendering. We can make use of this feature to render each segment of the terrain at the appropriate geometric level of details depending on distance.

In order to understand how this works, we need to take a look at dynamic tessellation in OpenGL [4]. Dynamic tessellation extends the OpenGL pipeline by 3 new stages located between the vertex and fragment shaders: a *tessellation control shader*, a fixed-function *primitive generation* and a *tessellation evaluation shader* (see Figure 3).

- The *tessellation control* shader is executed for each input vertex and gets the whole *patch* as an input. It decides how the edges and the interior of the patch are going to be tessellated.
- The resulting tessellation information is then passed as an input to the *primitive generation* stage which generates the final primitives (points, lines or triangles) for the requested tessellation levels.
- Then, the *tessellation evaluation shader* interpolates additional attributes (e.g., UV coordinates, etc.) for each newly created vertex of the tessellated primitives.

This high-level explanation will become much clearer once we apply this to terrain rendering. We start out by rendering a very coarse and uniform planar quad mesh for our terrain centered on the camera position. Unlike in previous assignments, the vertex shader does not perform any projection. It merely passes the vertices through to the tessellation control shader. This shader receives a complete quad as an input and determines the tessellation levels based on the distance of the patch to the camera. The closer the quad is to the camera, the higher its tessellation level required to capture the visible detail of the terrain. The primitive generation performs the requested tessellation and passes the generated geometry further down the pipeline to the tessellation evaluation shader. This shader computes the correct position and texture coordinates for each generated vertex. For this, it gets the coordinates passed in the abstract coordinate space of the original quad and interpolates the position on the ground plane and looks up the height in the height map. As this is the final shader stage before the rasterization, the screen space projection must also be performed in this shader[3].

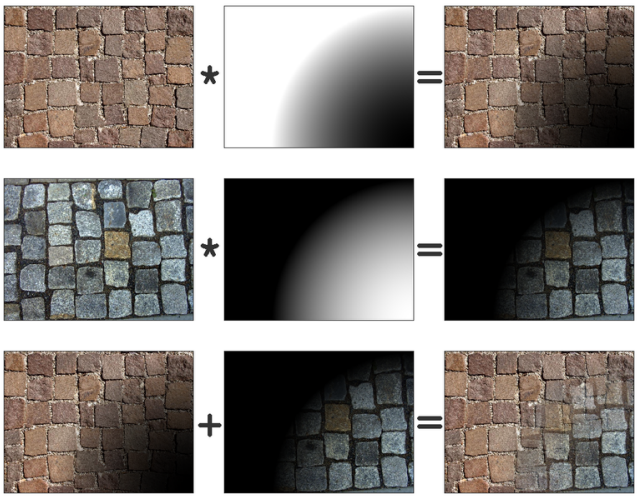

Figure 2: *An example of texture splatting with two facture textures and one modulating weight (the last weight is usually computed as* $1 - \Sigma(all\_weights)$*, i.e. as a remaining weight).*

2 Implementation

In this section we give more implementation details.

2.1 Integrating into the existing framework

Implement a new scene graph node for terrain in your modelling application, which will be able to render terrain height maps. The user should be able to change the horizontal and the vertical scale of the terrain. When the user adds a new terrain node to the scene, display a QFileDialog so he can select and open a height map. Additionally, the free-space camera movement (e.g. WSAD-like controls) can be implemented to fly along the huge terrain surface.

To make use of the tessellation shaders it is mandatory to create an OpenGL context with minimum version of 4.0. If you used a lesser OpenGL version for the previous assignments, you should create a compatibility profile which ensures that your previous code will continue to work. Since OpenGL 3.2, a lot of old functionality got deprecated. To store and render geometry using a GLSL4 shader you should thus use a *Vertex Array Object* [5]. Note that for a tessellation shader `GL_PATCHES` is the only valid input primitive. Furthermore the implicit declaration of uniform and varying variables in the shaders got removed, and you therefore need to explicitly define the input and output variables in your shaders. You also need to supply the transformation matrices manually using `glUnifom`. If you have trouble migrating to the newer OpenGL functionality, you can consult the *Tutorials for modern OpenGL* [2].

2.2 Loading textures

We provide you with the height map in PGM format and several tiled facture textures in JPEG format. PGM is a very simple and straight-forwards to read format. It stands for the **P**ortable **G**rayscale **M**ap. This format is widely supported, so you can open it in the majority of image authoring/viewing program like Photoshop or GIMP in order to check the data.

The PGM file consists of two parts:

- The human-readable header, where the image description is stored (dimensions, type, etc.)
- The binary image data (placed right after the header), which can be loaded directly into the chunk of system memory

The header consists of four lines:

- The first line contains two characters: "P5" - this is the type of image (P5 stands for the PGM binary image type).

[3] Additionally, a geometry shader can be invoked for each primitive after the tessellation evaluation shader, but we do not use it in this assignment.

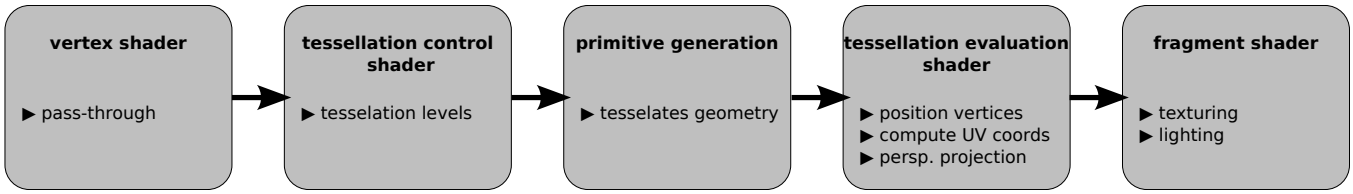

Figure 3: *Different stages of the hardware tessellation pipeline in a modern GPU.*

- The second line is the width of the image in ASCII format.
- The third line is the height of the image in ASCII format.
- The fourth line contains the maximum value of the image data in ASCII format. At a maximum value below 256 every the binary data is stored in 8 bits per channel, otherwise (< 65536) in 16 bits per channel. (Don't forget to read the last newline character after the value.)
- A raster of $Width \times Height$ grey values. The most significant byte is the first, therefore, if you simply load the values in a byte array, you have to swap the most and least significant bytes.

Loading a height map. The height map is stored in PGM format, which means that it is a one channel per pixel format for storing grayscale images.

The maximum value of the height is 65535, which can be stored using the 16-bit `unsigned short` type of C language. This means that you can allocate an array of unsigned shorts of the size $width \times height$ elements and load the binary data (which follows immediately after the header) into this chunk of memory.

The file provided for the height map is called *grand_canyon.pgm* and represents a part of the Grand Canyon located in North America scanned from the satellite. We won't exactly reconstruct the real world units, so you need to introduce the horizontal scaling parameter and the elevation scaling parameter. You should scale the map and the elevation until it looks visually plausible.

The size of the height map is 4096×4096 pixels, so you should create a GPU texture of that size with a complete mip pyramid and upload the data to the highest mip level of the texture. In OpenGL, we recommend to use `glTexImage2D` function for updating the content of the texture from CPU.

The final thing you should do with this height map is to generate the higher-level mip maps. Each mip level should be generated using a box filter (every pixel of the mip map is an average value of the four pixels of the finer parent level). We recommend using the GL function `glGenerateMipmap` for the automatic mip map generation.

Loading detailed facture textures. All facture textures are stored in JPEG format, which you can load using your Qt framework. We also encourage you to use your own textures for wider variety of materials.

2.3 Dynamic tessellation

The terrain geometry that is rendered should be centered around the camera. A grid of roughly 50×50 regular quads should be a good starting point. This geometry acts as a sliding square window on the height map. In case the position of the camera is changed, it should be moved with the camera in order to sample the proper elevation from the height map. This translation can be done in the vertex shader.

To render this grid you can either use the immediate mode of OpenGL or you can store the geometry in a *Vertex Buffer Object* during startup (see more details on it here [6] or in the course [3, slide 55]). This VBO can be then reused.

Now after we got a flat grid, we need to get it tessellated. For this, we need to create a tessellation control shader and a tessellation evaluation shader. We recommend that you start out with a simple tessellation control shader that sets fixed inner and outer tessellation levels and verify that the hardware tessellation is actually working by rendering the terrain in wireframe mode. Some aspects of the primitive generation must be specified with `layout` statements in the tessellation control and tessellation evaluation shaders. In the control shader you need to specify the number of vertices that you pass on to the primitive generation (4 in our case) with such a statement. The `layout` statement in the evaluation shader configures the output of the primitive generation. You should set it to work on `quads` as abstract patches and to emit the new vertices with *even spacing*. To position the new vertices, the evaluation shader receives the vertex attributes for the original patch as well as the coordinates of the new vertex in an abstract 2D coordinate space in the variable `glTessCoord`. The x and y coordinates go from 0 to 1 each. Use these coordinates to compute the new vertex position as a bilinear interpolation between the original quad corners.

Once this is working, you can add the computation of the tessellation factors based on the distance to the camera. Compute the distance between the patch center and the camera and chose a tessellation factor based on that. Functions like $\frac{1}{d^2}$ or e^{-d} with appropriate scaling provide good starting point. However, in order to render terrain properly, we need to restrict ourselves to the *equal_spacing* tessellation mode and tessellation levels which are powers of two. This allows us to improve the quality of the terrain rendering later on Experiment with your distance falloff function until you see a reasonable distance-based tessellation.

2.4 Sampling the height map

The loaded height map texture can be bound to the tessellation evaluation shader exactly the same way as to the fragment shader. However note that in the tessellation shaders there is no `texture` lookup available, since there is no way to correctly determine the mip level to lookup from. Thus you need to use `textureLod(texture, texCoord, mipLevel)` lookup method in order to perform a texture fetch in shaders other the fragment shader. The last value of this function is the sampled mip level. Zero means that we want to sample the finest mip level of the texture.

2.5 Improving height sampling and reducing jarring

At this point, the terrain rendering is not very convincing when the camera is moved around. You may notice that the terrain changes shape in quite unconvincing ways. We will now fix these issues in

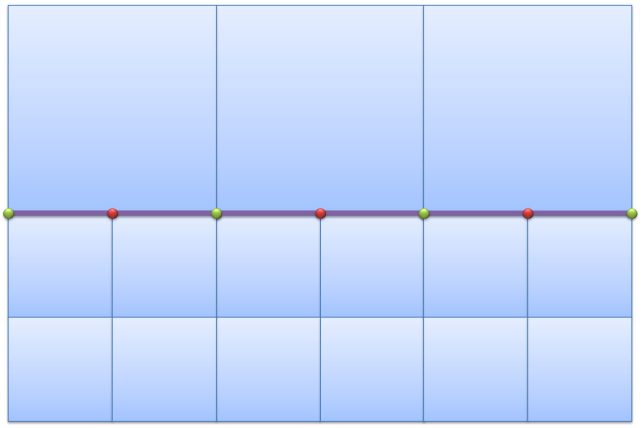

Figure 4: *Seams might occur on the border of two patches with different tessellation levels when using the same inner and outer tessellation levels for the quads. To fix this, the outer tessellation levels of adjacent patches should be the same so that vertices positions line up.*

two steps.

The terrain height is only sampled at vertex positions. When those glide smoothly over the terrain with the camera, some pronounced features like sharp edges or steep mountain tops will change shape drastically depending on where they are sampled. These changes can be reduced by snapping the geometry position to the height map in such a way that vertices always stay in the same position. In other words, you need restrict the shifting of the coarse geometry to exact multiples of the edge length of a quad in that mesh.

In order to further reduce irregular sampling of the height map during the camera movement, every level should be filled from the corresponding mip level of the height map. This provides a smooth filtering for the outer tessellation levels and reduces the geometry shivering at distance. The mip level should be selected such that the world-space cell size at the tessellation level should be the same as the world-space texel size of the selected mip level. So, the finest tessellation level you use precisely maps to a square subset of the finest mipmap level. The next tessellation level should sample the next mipmap level of the heightmap. In order to look up the corresponding mip level, we recommend to compute the binary logarithm of the tessellation level in the tessellation control shader and pass it on to the tessellation evaluation shader. Then you can simply use this value in the tessellation evaluation shader and look up the corresponding mip level of the height map, so the height can be sampled as follows: `textureLod(texture, texCoord, maxLogTessellationLevel - logTessellationLevel)`.

2.5.1 Removing seams between two tessellation levels

Since each finer tessellation level is connected to the coarser level, it is important to achieve a watertight connection at the transition. As illustrated in Figure 4, the seams might occur at the junctions of two quads of the finer level to the one quad of the coarser level (each finer level is twice denser). So the red vertex of the finer level that lies between two adjoint green vertices of the coarser grid might be displaced non-linearly comparing to the edge of the coarser quad. This usually produces an undesired seam at this place. In order to avoid such seams, we need to make sure that these common edges have the same tessellation. This can be achieved by setting the same outer tessellation levels on both patches. To do this, we suggest the following method: in the tessellation control shader, compute the tessellation levels for the neighboring patches as well and chose the lower or higher one consistently for each edge.

2.6 Terrain shading and illumination

After the geometry of the terrain surface is being successfully rendered, you can start implementing the shading variety and the proper illumination of the terrain surface. This consists of two steps:

1. Rendering the materials with the methods mentioned in Section 1.2
2. Illuminating the materials with light sources of the scene

These processes are independent from each other, so we can combine different material composition methods with different illumination techniques (Lambert/Phong model, shadows etc.).

2.6.1 Composition of materials

As discussed in 1.2, there are different methods of combining facture textures with each other. The mapping of the facture texture can be generated by scaling the pair of (x, z) coordinates of each vertex. The scaling should be chosen in a way that the facture textures look detailed enough when the camera approaches it closely. For this assignment we propose to implement a simple procedural material composition based on the combination of the height-based and slope-based mixing approaches. The slope-based approach should override the decision of the height-based approach on choosing the facture texture.

Height-based mixing. The height range is chosen for every facture texture. For example, for the height of $[0; 50)$ meters, the sand texture is chosen for texturing. If the height is in the range of $[50; 100)$ meters, then the rock texture is chosen instead. And so on. Also make sure to make a smooth transition between two tiled fractal textures on the borders of the height ranges, for example, using GLSL `smoothstep` function.

Slope-based mixing. The slope-based mixing is achieved by computing a mixing weight based on the slope of the surface. The simplest way of computing such weight is to take the y component of the normalized normal vector. The closer this component to zero the more vertical the slope is. You can find how to compute the normal from the height map in the Appendix A.

2.6.2 Illumination

You can reuse the simple Phong illumination model from the past assignment. Make sure you also use different specular brightness and glossiness exponents for different materials (e.g. you can make the rocks more shiny than the sand etc.).

3 Wrapping up your solution

It would be a plus to have a simple collision with camera. For example, the height is requested from the height map by the (x, z) coordinates of the camera position, then the camera position is adjusted such that it always stays above the terrain.

Set up the parameters for the terrain such that it covers a large area and looks like a realistic canyon in your framework.

Create a folder called results and place your screenshots there. Note that every screenshot should be coupled by its wireframe rendering, so we could see the tessellation working correctly. Do not post processes your screenshots using image editing software.

References

[1] Terrain LoD: Runtime regular-grid algorithms.
`http://vterrain.org/LOD/Papers`.

[2] Tutorials for modern opengl (3.3+).
`http://www.opengl-tutorials.org`.

[3] Carsten Dachsbacher. Computergraphik. kapitel 8: Opengl und grafik-hardware.
`http://cg.ibds.kit.edu/lehre/ws2010/cg/downloads/internal/08_OpenGL.pdf`.

[4] Khronos. Tessellation.
`http://www.opengl.org/wiki/Tessellation`.

[5] Khronos. Vertex array object.
`http://www.opengl.org/wiki/Vertex_Array_Object#Vertex_Array_Object`.

[6] Khronos. Vertex buffer object.
`http://www.songho.ca/opengl/gl_vbo.html`.

A Computing the normal

The normal of the surface at the point (x, y, z) can be computed as vector product of three adjacent cells of the height map:

$$\mathbf{N}(x,y,z) = \frac{[(\mathbf{H}(x+h,z) - \mathbf{H}(x,z)) \times (\mathbf{H}(x,z+h) - \mathbf{H}(x,z))]}{\| [(\mathbf{H}(x+h,z) - \mathbf{H}(x,z)) \times (\mathbf{H}(x,z+h) - \mathbf{H}(x,z))] \|}$$

where h is the step of the height map grid, $\mathbf{H}(x, z)$ - point in 3D space formed by (x, y, z), where y - is the elevation of the height map at (x, z).

Compute the normals in the tessellation evaluation shader.

Grading

You can achieve a total of 20 points in this assignment, based on the following criteria:

1 p	You implement a terrain node and provide a `QFileDialog` to load height maps and facture textures.
4 p	The terrain is displayed using quad patches and hardware tessellation, and can be both horizontally and vertically scaled.
1 p	The terrain tessellation changes with viewing distance in a reasonable manner.
2 p	The terrain geometry follows the camera correctly and can be correctly rendered in multiple viewports simultaneously.
2 p	The jarring of the geometry is eliminated by sampling the proper mip map level of the height map.
2 p	Terrain has no cracks nor seams at transitions between tessellation levels.
2 p	Camera collides with the terrain and never goes under it.
3 p	The material composition works nice with both the slope-based and the height-based weights and looks realistic.
2 p	The terrain is correctly illuminated by all light sources in the scene with Phong model.
1 p	The terrain has different illumination properties (e.g. specularity) for different materials.

You get a *malus* if your code is not nicely written using meaningful variable names in proper English, or significantly lacks proper comments! Moreover, you get a malus if your solution produces errors or way too many warnings during compilation! Lastly, you lose points if your program crashes or freezes for weird reasons.

If we got the impression that you handed in copy-pasted code, and you are unable to explain your solution, we will expel you from the course!