J. ANDREAS BÆRENTZEN, MORTEN NOBEL-JØRGENSEN, JEPPE REVALL FRISVAD, AND NIELS JØRGEN CHRISTENSEN

HANDS ON REAL-TIME GRAPHICS

Contents

1
Introduction

This book is an attempt at making a useful cross section through methods for real-time graphics. Rather than focus exclusively on the graphics pipeline itself, on the software architecture of a render engine or on any single, specific topic, we discuss a broad range of topics such as efficient rendering, procedural synthesis, deferred rendering, shadows, ambient occlussion, sky models, animation, volumetric scenes, and more.

On the other hand, while we aim for a relatively broad book, there are many things which we do not strive for: In particular, we do not cover all aspects or all literature about the discussed methods, and we do not attempt to teach the intricacies of a particular graphics API.

We do hope that someone who has mastered the methods covered in this book would be well poised for work on an "engine" for real-time graphics. We do believe such a person would be able to learn new methods in this field with relative ease.

Together with this book, an exercise platform has been made available. The exercises in this book are based on the interactive scenes provided by the platform, and in the exercise instructions references are made to files in the framework of the platform. While the main text of the book is intended to be readable even without the exercise platform, the exercise instructions are not.

Enjoy graphics programming!

2
The Real-Time Graphics Pipeline

Computer graphics is about visualization, which we often call *rendering*, of 3D models. For some applications, it is not essential that this rendering is *real-time*, but if user interaction is involved, it is important that the user gets immediate feedback on her input. For fast paced computer games up to around 60 Hz may be required, but in many cases we can make do with somewhat less. Arguably, the word "real-time" means just that the clock used in the simulation (or computer game) is neither slower nor faster than a real world clock. In other words, there is no precise definition of how many frames per second it takes before rendering is real-time. Nor is there any prescribed method for real-time rendering. However, apart from some servers or the cheapest netbooks, recent computers invariably include hardware dedicated to real-time rendering. Such hardware is usually an implementation of the pipeline described in this note. In PCs the hardware is normally in the form of a graphics card which contains a *graphics processing unit* (GPU) as the central processor. Recent GPUs include some of the most powerful (in terms of operations per second) chips ever made.

A graphics card is simply a machine for drawing triangles with texture. Of course, a graphics card is also capable of drawing other *primitives* such as general polygons, points, and lines, but triangle drawing and texture mapping are the essential features.

From the programming point of view, we need a driver and an API which allows us to send data and commands to the graphics card. While this text makes only few references to the notion of a graphics API, it is important to mention that there are several possible choices. On the windows platform Direct3D remains the default choice, on Apple platforms the more recent Metal API is an obvious way to go, on Android based platforms and the web, we would normally choose GLSL. On Linux, OpenGL continues its reign albeit the more recent version called Vulkan is very different from classic OpenGL and also seems to require much more work to get started

than previous OpenGL versions.

Indeed it seems the world of graphics APIs has been divided into two. More and more the casual users will probably use very high level APIs whereas game developers will use thin APIs that leave much work to the developers. It seems a little odd that it should be so hard to develop a powerful API that could be the weapon of choice for both the casual users and more advanced users, but this is the situation we are in.

The goal of this chapter is to give you a fundamental, functional understanding of how the real-time rendering pipeline of a graphics card works. "Functional" means that we see things from the programmers point of view rather than the hardware designers. We do not avoid the mathematics behind graphics but try to explain things in the most concise fashion possible.

2.1 *Overview of the pipeline*

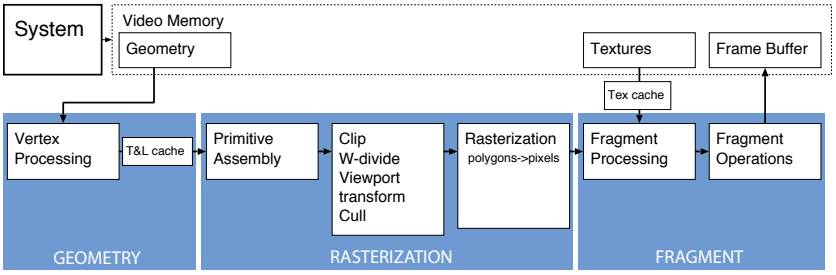


Figure 2.1: The pipeline in a graphics card. We can roughly divide the pipeline into a geometry part where vertices are transformed, rasterization where triangles are turned into fragments (potential pixels), and a fragment part where we process fragments and finally write them to the framebuffer.

The pipeline of a graphics card is illustrated in Figure 2.1. The input is geometry in the form of triangles, textures, and graphics commands. The corners of triangles (and other geometric primitives) are denoted *vertices*. Generally, we associate more attributes than just a geometric position with a vertex. To give a few examples, we often associate color, texture coordinates, and a normal vector with a vertex. As shown in the Figure, the first thing that happens is that we compute lighting (i.e. color) for the vertices and transform them as described in the next section. After lighting and transformation, which we collectively denote *vertex shading*, we assemble the primitives (triangles) and perform a number of steps: We *clip* away geometry that is outside the *viewing frustum*, i.e. the part of the world space that maps to our screen. We often *cull* (i.e. remove from the pipeline) triangles which face away from the camera, and we perform the final part of the perspective projection which is the w divide. Triangles

are then rasterized which means they are turned into *fragments*: Potential pixels that have not yet been written to the framebuffer are denoted fragments. Next, the color is computed for each fragment. In the simplest cases, this process, known as *fragment shading*, is simply an interpolation of the color between the vertices of the triangle. After shading, further fragment operations, such as *depth testing*, are performed before the fragment is written to the framebuffer.

Note that the above process is easy to implement in a parallel fashion. The computations on a vertex are independent from those on any other vertex, and the computations on a fragment are independent from those on any other fragment. Consequently, we can process many vertices and fragments in parallel, and, in fact, this parallelism is well exploited by modern graphics processing units. Note also that this is the classical pipeline. As we shall learn later, the processing of both vertices and triangles is now programmable, and there are also programmable stages which are not shown in the figure.

2.2 *Vertex Transformation and Projection*

An important part of the graphics pipeline is the geometric transformation of vertices. We always represent the vertices in homogeneous coordinates. This means that a 3D point is specified in the following way

$$\mathbf{p} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

where w is almost always 1. Note that letters in boldface denote vectors (or matrices). Note also that vectors are column vectors unless otherwise stated. Just as for vectors, we operate with matrices in homogeneous coordinates. In other words, the matrices we use are of the form

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

As you see above, we use capitals in boldface to denote matrices. Since we use column vectors, we multiply vectors onto matrices from the right:

$$\mathbf{q} = \mathbf{M}\mathbf{p}$$

For more details on linear algebra, please see the Appendix A in the course notes [1].

[1] Henrik Aanæs. *Lecture Notes on Camera Geometry*. DTU Informatics, 2009

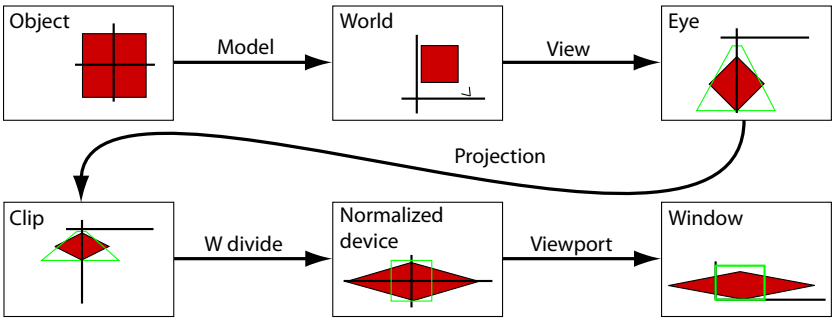


Figure 2.2: The coordinate systems used in real-time rendering. The words in the boxes denote coordinate systems, and the words over the arrows denote transformations. The red box is the object we render, and the coordinate axes are indicated by heavy black lines. The viewing frustum of the virtual camera is indicated by a green box.

A number of coordinate systems are used in real-time rendering, and to a large extent the goal of this section is to give you a fairly precise understanding of why we need these coordinate systems and what transformations take you from one to the next (cf. Figure 2.2).

First of all, you specify the vertices of a 3D object in a coordinate system which is convenient for the object. Say you model a rectangular box. In this case you would typically align the edges of the box with the axes of the coordinate system and place its center in the origin (cf. Figure 2.2). However, this box needs to be transformed in order to place and orient it in the scene. This is often called the *model transformation*. After the model transformation, your box is in *world coordinates*.

In computer graphics, we often use a special coordinate system for the camera, and it is more convenient to think about the camera transformation as a transformation which takes the scene and positions it in front of the camera rather than a transformation which positions the camera in front of the scene. Consequently, we will use the interpretation that your box is next transformed into *eye coordinates* - the coordinate system of the camera, which, incidentally, is always placed at the origin and looking down the negative Z axis. The coordinate system is right handed: if we look in the direction that the camera points, the X and Y axes point left and up, respectively, and the Z axis pops out of the image. The transformation is called the *view transformation*.

In eye coordinates, we multiply the vertices of our box onto the projection matrix which produces *clip coordinates*. We perform the so called *perspective divide* to get *normalized device coordinates* from which the *viewport transformation* finally produces *window coordinates*. The coordinate systems and transformations are illustrated in Figure 2.2.

In the rest of this section, we will describe this pipeline of transformations in a bit more detail. Since it is a programmable pipeline, we are just describing one way that the transformations could be done, and not the definitive way. That being said, we are describing the common approach. However, in some cases, we may already have world coordinates and the model transformation is not needed. In other cases, we may have 2D data where even the view transform is unnecessary.

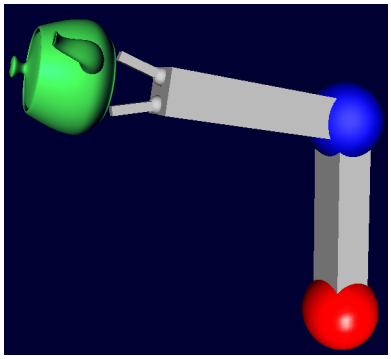


Figure 2.3: A robot arm created with simple primitives (apart from the teapot). Each primitive has been transformed from object to world coordinates with its own model transformation.

2.2.1 *Model Transformation*

This first transformation assumes that our objects are not directly represented in the world coordinate system. World coordinates is the system where we represent our scene, and normally we have a separate coordinate system for each of the objects that go into the scene. That is convenient because we might have many *instances* of the same object in a scene. The instances would only differ by model transformation and possibly the attributes used for rendering. For example, observe the robot arm "holding" a teapot in Figure 2.3. The scene contains four cubes, four spheres and a teapot. The cubes and spheres are all drawn via calls to the same function in the graphics API but with different model transforms.

In principle, we can use any 4×4 matrix to transform our points from object to world coordinates. However, we generally restrict ourselves to rotation, translation and scaling (and sometimes reflection). We will briefly show how the corresponding matrices look in homogeneous coordinates. 3×3 rotation matrices for 3D rotation are described in details in Appendix 3.1 of [2]. In homogeneous coordinates, the 3×3 rotation matrix, $\vec{R}^{3\times3}$ is simply the upper left 3×3 matrix of a 4×4 matrix:

[2] Henrik Aanæs. *Lecture Notes on Camera Geometry*. DTU Informatics, 2009

$$\vec{R} = \begin{bmatrix} & & & 0 \\ & \vec{R}^{3\times3} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The important thing to note is that the w coordinate is unaffected. The same is true of a scaling matrix which looks as follows

$$\vec{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where s_x, s_y, and s_z are the scaling factors along each axis. Finally, a translation matrix looks as follows

$$\vec{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\vec{t} = [t_x\ t_y\ t_z]^T$ is the translation vector.

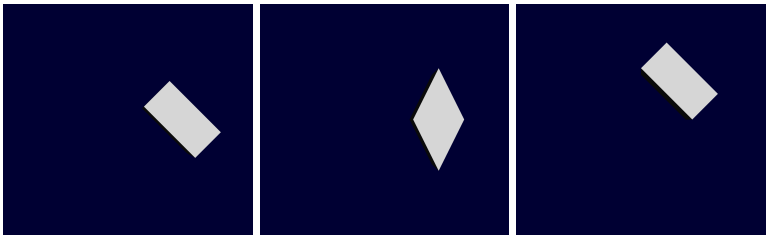


Figure 2.4: These three images show the significance of the order of transformations. To the left, the box has been scaled, rotated, and translated. In the middle it has been rotated, scaled, and translated. To the right it has been translated, scaled, and rotated. The individual transformations are the same in all three cases. The scaling scales one axis by 2.0, the rotation rotates by 45 degrees, and the translation is by 1.5 units along one axis.

Transformations are concatenated by multiplying the matrices together, but matrix multiplication does not commute. This means that the result is not invariant with respect to the order of the transformations. For instance, scaling followed by rotation and finally translation is not the same as translation followed by rotation and then scaling – even if the individual transformations are the same. Normally, we first scale the object and then rotate it and finally translate it. Using a different order typically leads to surprising and unwanted results, but sometimes we do need a different order. The importance of order of transformations is shown in Figure 2.4.

The model transformation is unique in that it changes per object. Every object needs its own model transform whereas the entire image is almost always drawn with just one view transform (described in the next section) and one type of projection.

We often need to transform in a hierarchical fashion. One way of understanding this is that we could use one set of model transformations to put together a few primitives into a new object which is then transformed into world space with a second model transformation. For instance, the upper and lower arm as well as the two fingers of the robot in Figure 2.3 are nearly identical compositions of a sphere and a box. Thus, we could create a robot-arm segment object by scaling and translating a cube and a sphere and then, subsequently, create four instances of this object, scaling, rotating and translating each to the proper position.

2.2.2 *View Transformation*

The view transformation is a translation followed by a rotation. The translation moves the scene so that the camera is at the origin, and the rotation transforms the centerline of the projection into the negative Z axis. This rotation could in principle be computed by a composition of basic rotations, but is much more convenient to simply use a *basis change matrix*.

Say the user has specified the camera location, $\vec{e}$, the direction that the camera points in (i.e. line of sight), $\vec{d}$, and an up vector, $\vec{u}$. The up vector points in the direction whose projection should correspond to the screen Y axis. See Figure 2.5.

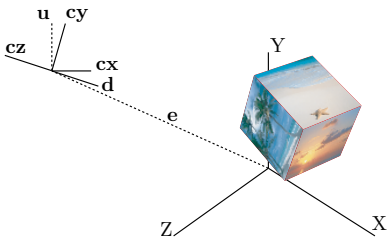


Figure 2.5: The vectors $\vec{d}$, $\vec{e}$, and $\vec{u}$ which are needed to compute the basis (in world coordinates) for the eye coordinate system as well as the basis itself ($\vec{c^x}$,$\vec{c^y}$,$\vec{c^z}$)

Based on these three vectors, we need to compute the translation and rotation of the camera. The camera translation is simply $-\vec{e}$ since translating along this vector will move the camera to the origin. To compute the rotation, we need to compute a basis for the eye coordinate system. This basis will be formed by the three vectors $\vec{c^x}$, $\vec{c^y}$, and $\vec{c^z}$. The last one is fairly easy. Since the camera looks down the negative Z axis, we have that

$$\vec{c^z} = -\frac{\vec{d}}{\|\vec{d}\|} \tag{2.1}$$

We need to compute a $\vec{c^x}$ so that it will correspond to the window X axis, which means that it should be orthogonal to the window Y axis and hence the up vector:

$$\vec{c^x} = \frac{\vec{d} \times \vec{u}}{\|\vec{d} \times \vec{u}\|} \tag{2.2}$$

Finally, $\vec{c^y}$ should be orthogonal to the two other vectors, so

$$\vec{c^y} = \vec{c^z} \times \vec{c^x} \tag{2.3}$$

These three vectors have length 1. Now, we can write down the viewing transformation

$$\vec{V} = \begin{bmatrix} {c^x}_x & {c^x}_y & {c^x}_z & 0 \\ {c^y}_x & {c^y}_y & {c^y}_z & 0 \\ {c^z}_x & {c^z}_y & {c^z}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Note that we use only a single modelview transformation matrix which we will denote $\vec{MV}$. In other words, the matrices for model transformation and view transformation are multiplied together. This makes sense because we need the vertices of the object we are rendering in eye coordinates (after model and view transformation)

since this is where we compute lighting. However, we rarely need the points in world coordinates and saving a matrix vector multiplication for all vertices can be a big advantage.

2.2.3 *Projection*

If we think of computer graphics as rendering a virtual scene with a virtual camera, we obviously think of projections as mappings from a 3D world onto a 2D image plane. However, we can perform two different kinds of projections: Orhtographic and perspective. In orthographic projections, the *viewing rays* which connect a point in space to its image in the image plane are parallel and also orthogonal to the image plane. In perspective, there is a center of projection, and all viewing rays emanate from that point. In fact, there is a third type of projection, which we will not cover in detail, namely *oblique projections* where the rays are parallel but not orthogonal to the image plane. The various type of projections are illustrated in Figure 2.6.

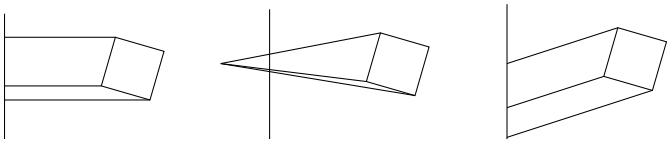


Figure 2.6: From left to right, this figure illustrates orthographic, perspective, and oblique projections.

While projections in a sense do reduce dimension, this happens only when we throw away the Z value, and, in fact, we often care a lot about the Z value in computer graphics, not least because the Z value (or depth) is used to depth sort fragments as discussed in Section 2.3.4. Thus, it makes sense to see the projections (parallel or perspective) as mappings from a *view volume* in eye space to a volume in normalized device coordinates (NDC). The eye space volume is either a rectangular box if we are doing parallel projection or a pyramid, a *frustum*, with the top cut off if we are doing perspective. In either case, the view volume is delimited by six planes which are known as the near, far, top, bottom, left, and right clipping planes.

While the volume in eye coordinates can have different shapes, the projection always maps it into a cube of side length two centered at the origin. In other words, normalized device coordinates are always in the same range.

Orthographic Projection

We can express an orthographic projection using the following matrix

$$\mathbf{O} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where r, l, t, b, f, n are the maximum and minimum X, Y, and Z values denoted right, left, top, bottom, far, and near. The result of multiplying an eye space point onto $\mathbf{O}$ is a point in normalized device coordinates. Observe that if $\vec{p}_e = [r\ t\ -f\ 1]^T$, the point in normalized device coordinates is

$$\vec{p}_n = \vec{p}_c = \mathbf{O}\vec{p}_e = [1\ 1\ 1\ 1]^T$$

and likewise for the other corners of the view volume. Hence, it is easy to verify that this matrix does map the view volume into a cube of side length two centered at the origin. Note that there is no difference between clip and normalized device coordinates in this case since $w = 1$ both before and after multiplication onto $\vec{O}$.

It may be surprising that Z is treated differently from X and Y (by flipping its sign) but remember that we are looking down the negative Z axis. However, we would like Z to grow (positively) as something moves farther away from the camera. Hence the sign inversion.

Why look down the negative Z axis in the first place? The answer is simply that this is necessary if we want the normal right handed coordinate system in window space to have the familiar property that the Y axis points up and the X axis points to the right. For an illustration, please refer to Figure 2.7.

After parallel projection, the final step is viewport transformation which is described in Section 2.2.4.

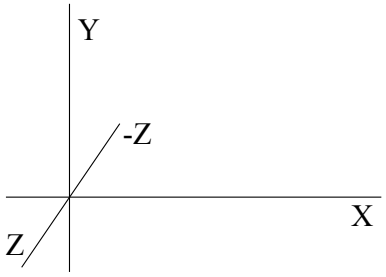


Figure 2.7: A normal right handed coordinate system. Note that we need to look down the negative Z axis if we want the Y axis to point up and the X axis to point to the right.

Perspective Projection

In perspective, things far away appear smaller than things which are close. One way of looking at this is that we need to compress the part of the view volume which is far away. Since the projection always maps the view volume into a cube, this gives an intuitive explanation of why the view volume is shaped like a pyramid with the apex in the eye and its base at the far plane as shown in Figure 2.8.

In computer graphics, we often use the following matrix for per-

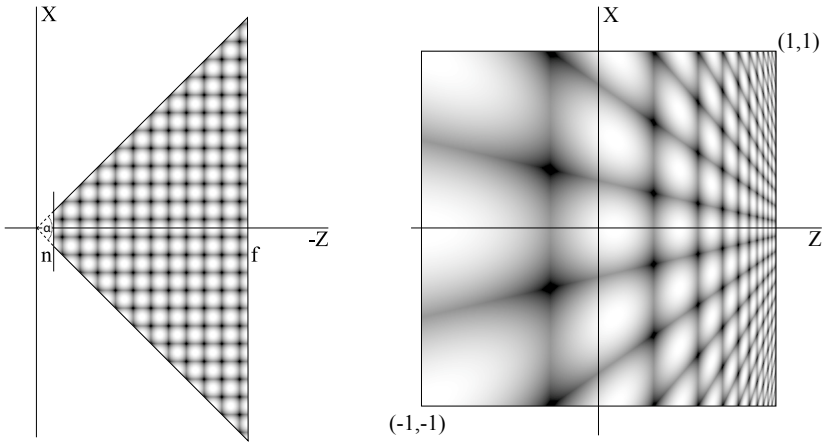


Figure 2.8: This 2D figure illustrates the perspective projection (including perspective divide). The view frustum is shaped like a pyramid with the top cut off at the near plane n and the bottom cut at the far plane f. The field of view angle, α determines how pointy the frustum is. The perspective projection maps the pyramidal frustum to a cube of side length two, centered at the origin. Note that as the frustum is transformed into a cube, the regular pattern is skewed, but straight lines map to straight lines.

spective projection

$$\mathbf{P} = \begin{bmatrix} \frac{1}{A}\cot\frac{\alpha}{2} & 0 & 0 & 0 \\ 0 & \cot\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.5}$$

where A is the aspect ratio, α is the field of view angle (in the Y direction), and n and f are the near and far clipping planes, respectively. The terms are illustrated in Figure 2.8 except for A which is the ratio of the width to the height of the on screen window. An important message, which can be gleaned from Figure 2.8, is that the precision of the depth buffer is far from uniform (observe how compressed things are close to the far plane): the precision is much higher close to the eye point than far from it. Moreover, the closer n is to zero, the worse the precision of the depth buffer overall. Careful graphics programmers strive to push n as far from the eye as possible and bring f as close as possible since this gives the best precision.

Now, we compute clip coordinates by

$$\vec{p}_c = \vec{P}\vec{p}_e$$

Note that $\vec{P}$ does map the view volume into a unit cube in homogeneous coordinates, but w is different from 1 for points in clip coordinates, so it is only after w division that the corners *look like* the corners of a cube. Meanwhile, we first perform clipping.

Clipping

After multiplying a point onto the projection matrix it is in clip coordinates. Not surprisingly, this is where clipping occurs. If our trian-

gle is entirely inside the view volume it is drawn as is. If it is outside, it is discarded. However, if the triangle intersects the view volume, we need to clip it to the volume. Points inside the view volume fulfill the inequalities

$$\begin{aligned} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ -w_c \leq z_c \leq w_c \end{aligned} \tag{2.6}$$

If one divides these inequalities with w_c it becomes clear that this is simply a test for inclusion in the NDC cube performed in homogeneous coordinates.

W divide

The final step of the perspective projection is the w divide which takes the point from clip to normalized device coordinates.

$$\mathbf{p}_n = \frac{1}{w_c}\mathbf{p}_c = \frac{1}{w_c}\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

Thus, the full perspective transformation consists of first multiplying eye coordinate points onto $\mathbf{P}$ and then performing the perspective divide. This full transformation is illustrated in Figure 2.8. Note that although it is is not linear due to the division, straight lines are transformed into straight lines, and the pyramidal view volume is transformed into the same cube as the view volume in an orthogonal transformation.

2.2.4 *Viewport Transformation*

The viewport transformation takes points from normalized device coordinates to window coordinates. Given a window of dimensions $W \times H$, the viewport transformation is simply a scaling and a translation. It can be written as a matrix, but normally we just write it directly:

$$\mathbf{p}_p = \begin{bmatrix} W\frac{x_n+1}{2} \\ H\frac{y_n+1}{2} \\ \frac{z_n+1}{2} \end{bmatrix} \tag{2.7}$$

After viewport transformation, the point is in coordinates which correspond to a point in pixel coordinates inside the *framebuffer*. Thus, if we have three vertices in window coordinates, we are ready to assemble the corresponding triangle and *rasterize* it.

2.3 *Triangle Rasterization and the Framebuffer*

Triangle rasterization is the process of turning triangles into pixels. This is arguably the most important function of a graphics card. The basic principle is simple: For a given triangle, we need to find the set of pixels that should represent the triangle in the screen window. The pixels are written to the so called framebuffer, which is simply an area of the graphics card memory that is used for intermediate storing of images which later get displayed on the monitor. A basic framebuffer contains a color buffer with red, green, and blue *channels* each of, typically, eight bit. This means that the color of a pixel is represented by 24 bits for a total of more than 16 million different colors which is usually sufficient. However, we often need an additional alpha channel in the color buffer which may be used to indicate transparency. In addition to the color buffer, we frequently need a depth buffer to resolve whether an incoming fragment is in front of or behind the existing pixel. Additional buffers are sometimes needed. For instance a *stencil buffer* can be used to mark regions of the window which should not be overwritten. The number of buffers and the number of bits per pixel depends on the mode of the graphics card, and usually graphics cards support a range of modes.

2.3.1 *Rasterization*

It is important to emphasize that a pixel is drawn if its center is inside the triangle. Confusingly, in window coordinates, the pixel centers are not the integer positions, say $[72\ \ 33]$. This is because we divide the window space into little pixels which can be regarded as squares. If the window lower left corner is at $[0\ \ 0]$ and pixels are of unit side length, it is clear that their centers are at the grid points in a grid which is shifted by half a unit in the X and Y directions. Thus, using the same example, the point we would check for inclusion in a triangle is $[72.5\ 33.5]$.

A simple way of rasterizing a triangle would be to use the interpolation methods described below to check for each pixel whether it is included in the triangle. A simple optimization would be to check only pixels inside the smallest rectangle containing the triangle. Note that graphics cards use faster and highly optimized methods which are furthermore implemented in hardware. The details of how rasterization is done are not generally known, but one important yet basic optimization is to use coherence: We can precompute a number of parameters which then do not have to be computed in the inner loop of the rasterization. In the context of triangle rasterization, this is called *triangle setup*.

However, testing whether a pixel is inside a triangle is not all: We also need to find the color for each pixel. As mentioned, this is often called fragment *shading*, and it can be done in many ways. However, the simple solution is that we first shade the vertices – i.e. compute a color per vertex. In fact this is done rather early in the pipeline in the vertex processing, but we will discuss shading later. Presently, we simply assume that we know the color per vertex and need to find the color per pixel. We do so by, for each pixel, taking a weighted average of the vertex colors where the weight for each vertex depends on the proximity of the pixel to that vertex. This is called *interpolation* and is the topic of the next two subsections.

2.3.2 *Interpolation of Attributes*

We typically have a number of attributes stored per vertex - for instance a vertex color, normal, or texture coordinates. Informally, we want to set the pixel color to a weighted average of the vertex colors where the weight of a vertex depends on how close the pixel is to the vertex (in window coordinates).

In practice, we always use linear interpolation to obtain the pixel values of some attribute from the values at the vertices. In 1D, linear interpolation is very easy, especially if the data points are at unit distance. Say, we have two 1D vertices p_0 and p_1 at unit distance apart and a point q on the line between them. We wish to interpolate to q as illustrated in Figure 2.9. It should be clear that if $\alpha = q - p_0$ then

$$f = (1 - \alpha)f_0 + \alpha f_1$$

interpolates the value in a linear fashion, i.e. the interpolated values lie on a straight line between the two data points. We can write this in a somewhat more general form

$$f = \frac{p_1 - q}{p_1 - p_0} f_0 + \frac{q - p_0}{p_1 - p_0} f_1$$

which takes into account that the points may not be at unit distance.

Now, if we interpolate in a 2D domain (the triangle) the data points should no longer lie on a line but in a plane. Otherwise the setup is similar. Assume that we have a triangle with vertices labelled 0, 1, and 2. The corresponding 2D window space points are $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$, the attributes we wish to interpolate are f_0, f_1, and f_2. The function

$$A(\vec{p}_0, \vec{p}_1, \vec{p}_2) = \frac{1}{2}(\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)$$

computes the signed area of the triangle given by the points $\vec{p}_0, \vec{p}_1$, and $\vec{p}_2$. Note that $\times$ is the cross product of 2D vectors in this case,

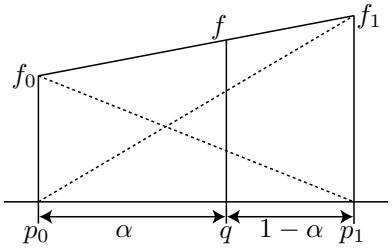


Figure 2.9: Linear interpolation between points p_0 and p_1 on the real line. The dotted lines show $(1 - \alpha)f_0$ and αf_1 whereas the solid line connecting (p_0, f_0) and (p_1, f_1) is the sum $((1 - \alpha)f_0 + \alpha f_1)$ and the line on which the interpolated values lie.

i.e. a determinant. According to this definition, the area is positive if the vertices are in counter clockwise order and negative otherwise. Finally, the point to which we wish to interpolate (the pixel center) is denoted $\mathbf{q}$.

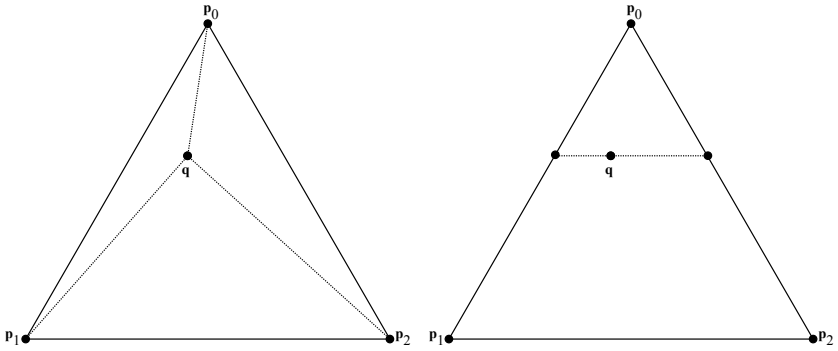


Figure 2.10: This figure illustrates the triangles and points involved in computing barycentric coordinates on the left. On the right an alternative scheme for linear interpolation where we interpolate using 1D linear interpolation to intermediate points on the same horizontal line and then interpolate to $\mathbf{q}$ along the horizontal line.

We can now compute the three so called *barycentric coordinates*

$$\mathbf{b} = [b_0, b_1, b_2]^T = \frac{[A(\mathbf{q}, \mathbf{p}_1, \mathbf{p}_2),\ A(\mathbf{p}_0, \mathbf{q}, \mathbf{p}_2),\ A(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})]^T}{A(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)} \tag{2.8}$$

Interpolating the f quantity (e.g. pixel color), we simply compute

$$f_{\mathbf{q}} = b_0 f_0 + b_1 f_1 + b_2 f_2 \tag{2.9}$$

As we can see from (2.8), the barycentric coordinates must sum to 1. In other words, we can compute $b_2 = 1 - (b_0 + b1)$.

If $\mathbf{q}$ is inside the triangle, the barycentric coordinates are always positive. If $\mathbf{q}$ is outside the triangle, the barycentric coordinates still sum to 1, but now the vertices of at least one of the triangles in Figure 2.10 are not in counter clockwise order, and the corresponding area(s) become negative. In other words, if the pixel center is outside the triangle, at least one of the barycentric coordinates is < 0.

We could also interpolate linearly in other ways. For instance, we could interpolate to two intermediate points along the $\mathbf{p}_0\mathbf{p}_1$ and $\mathbf{p}_0\mathbf{p}_2$ edges using 1D linear interpolation and then do another linear interpolation along the line segment between the two intermediate points to the final location. this scheme is also illustrated in Figure 2.10. Assuming a row of pixels lie on the horizontal line, this scheme could be more efficient than using barycentric coordinates.

Never the less, barycentric coordinates are a very general and useful tool in computer graphics and also in image analysis. Often we have data associated with vertices of a triangle, and we wish to interpolate this data. Barycentric coordinates are not just for triangles but

more generally for simplices. In a given dimension, a *simplex* is the simplest geometric primitive that has any area (or volume). In 1D it is a line segment, and in fact linear interpolation as described above is simply the 1D variant of interpolation with barycentric coordinates. In 3D, we can use barycentric coordinates to interpolate between the four vertices of a tetrahedron.

2.3.3 *Perspective Correct Interpolation*

Unfortunately, there is a problem when we see things in perspective. The simplest possible example, a line segment in perspective, is shown below.

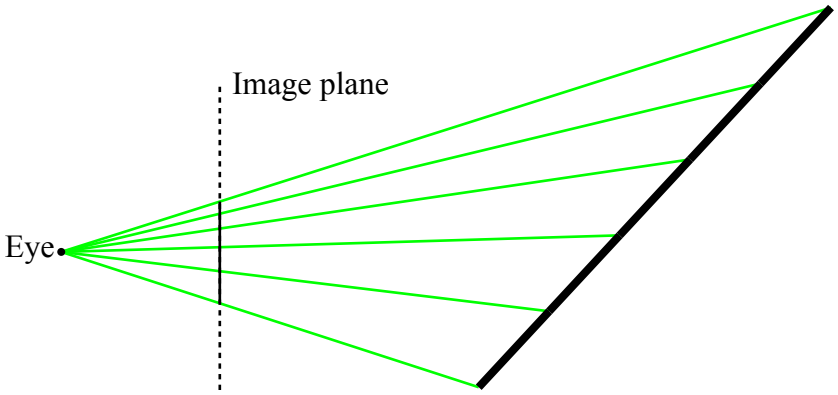


Figure 2.11: A line in perspective. The line is divided into equal segments, but these equal segments do not correspond to equal segments in the image. Thus, linear interpolation in object coordinates and image coordinates does not produce the same results.

Put plainly, stepping with equal step length along the 3D line does not correspond to taking equal steps along the line in the image of the line. If we fail to interpolate in a perspective correct fashion (simply use screen space linear interpolation), the result is as seen in Figure 2.19.

To perform perspective correct interpolation, the formula we must use is

$$f_{\mathbf{q}} = \frac{b_0 \frac{f_0}{w_0} + b_1 \frac{f_1}{w_1} + b_2 \frac{f_2}{w_2}}{b_0 \frac{1}{w_0} + b_1 \frac{1}{w_1} + b_2 \frac{1}{w_2}} \quad . \tag{2.10}$$

In the following, we shall see why.

The Details of Perspective Correct Interpolation

To perform perspective correct linear interpolation, we need to first express linear interpolation in eye coordinates (before perspective projection) and then compute what the eye space interpolation weights should be in terms of the window space weights. That is

not completely trivial, however, so to simplify matters, we will only consider the simplest possible case which is shown in Figure 2.12

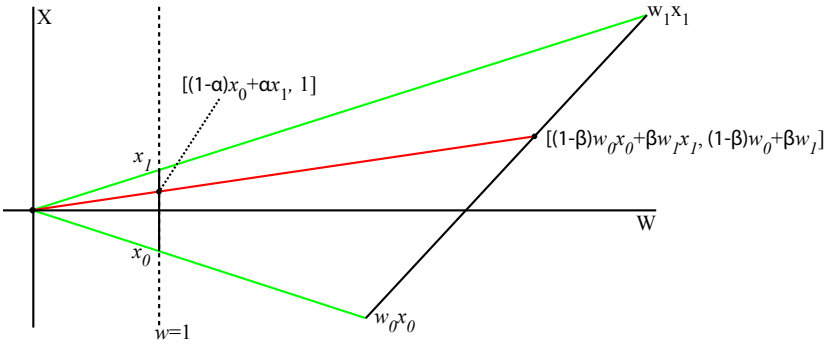


Figure 2.12: A line in perspective. The line is divided into equal segments, but these equal segments do not correspond to equal segments in the image. Thus, linear interpolation in object coordinates and image coordinates does not produce the same results.

We consider only the X and W axes and observe what happens precisely at the perspective division which takes us from clip coordinates (CC) to normalized device coordinates (NDC). We perform linear interpolation in both types of coordinates. The weight is α in NDC and β in CC. Given a point on a line in CC and its projected image in NDC, we want to find the equation which expresses β in terms of α. We start by writing down the equation which links the point in CC and its projection in NDC:

$$(1-\alpha)x_0 + \alpha x_1 = \frac{(1-\beta)x_0w_0 + \beta x_1w_1}{(1-\beta)w_0 + \beta w_1}$$

The interpolation can also be written

$$\alpha(x_1 - x_0) + x_0 = \frac{\beta(x_1w_1 - x_0w_0) + x_0w_0}{\beta(w_1 - w_0) + w_0}$$

Moving x_0 to the other side, multiplying it with the denominator in order to have just one fraction, removing terms that cancel, and reordering, we finally get:

$$\alpha(x_1 - x_0) = \frac{\beta w_1(x_1 - x_0)}{\beta(w_1 - w_0) + w_0}$$

We now divide by $(x_1 - x_0)$ and solve for β. Rewriting, we obtain

$$\beta = \frac{\alpha w_0}{w_1 - \alpha(w_1 - w_0)} \tag{2.11}$$

Now, we are nearly done. Say we have some quantity, f, associated with the end points that we want to interpolate to the given point. Linearly interpolating in CC amounts to

$$f = \beta(f_1 - f_0) + f_0$$

Plugging in (2.11),

$$f = \frac{\alpha w_0}{w_1 - \alpha(w_1 - w_0)}(f_1 - f_0) + f_0$$

which is straight forward to rewrite to

$$f = \frac{(1-\alpha)\frac{f_0}{w_0} + \alpha\frac{f_1}{w_1}}{(1-\alpha)\frac{1}{w_0} + \alpha\frac{1}{w_1}} \ .$$

What this mathematical exercise shows us is that to interpolate in a perspective correct fashion, we need to first divide the data that we want to interpolate with the w values at the corresponding vertices and we need to divide the interpolated value with the linearly interpolated inverse w values. This scheme also works for interpolation with barycentric coordinates, and it is now possible rewrite (2.9) to take perspective into account

$$f_{\mathbf{q}} = \frac{b_0\frac{f_0}{w_0} + b_1\frac{f_1}{w_1} + b_2\frac{f_2}{w_2}}{b_0\frac{1}{w_0} + b_1\frac{1}{w_1} + b_2\frac{1}{w_2}} \ .$$

The above equation is used to interpolate almost all vertex attributes and it is particularly important for texture coordinates. The one exception is depth. A triangle in object coordinates maps to a (planar) triangle in window coordinates. Consequently, the window coordinate Z values can be linearly interpolated over the interior of the triangle with no need for perspective correction, and it is indeed the linearly interpolated window coordinate Z value which is stored in the depth buffer. The depth buffer is covered in more detail in the next section.

2.3.4 Depth Buffering

When we are ready to write the color and depth value of a fragment to the framebuffer, there is one test which we nearly always want to perform, namely the depth test.

The result of enabling and disabling the depth test is shown in Figure 2.13. Without the depth test it is clear that parts of the cube which are visible have been covered by parts which should not have been visible. Without depth testing, the pixels which we see are those that have been drawn last.

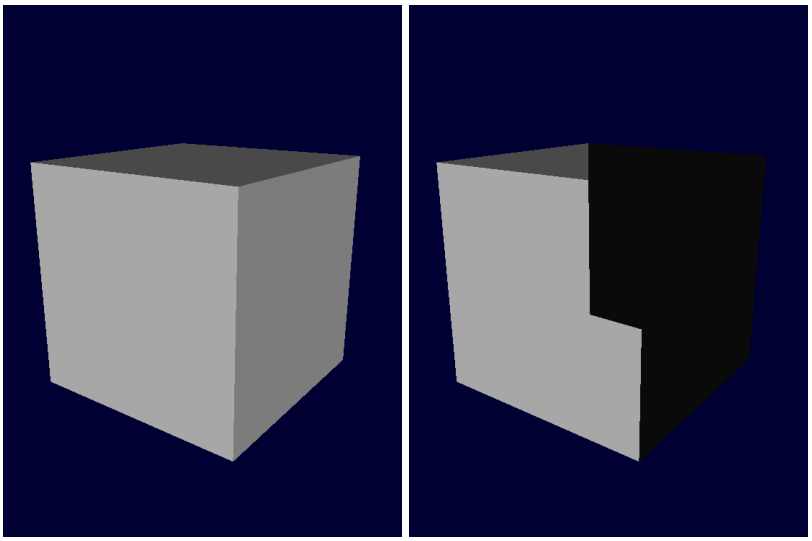


Figure 2.13: The result of drawing a box with depth test (on the left) and without depth test (on the right).

In this particular case, although depth testing solves the problem, it is actually not needed: We could simply cull the faces which point away from the camera, and graphics hardware will do that for us. However, for bigger scenes or just objects which are not *convex*, we do need depth testing unless we draw the model in strict back to front order. That requires us to sort the triangles and deal with the tedious case of intersecting triangles and in practice, we rarely go that way.

Depth testing works well for most purposes, but one issue is the fact that the precision is not linear. This is easy to understand when we remember that it is the window space depth value which is stored and not the eye space depth value. Window space depth is just a scaling of normalized device coordinates depth which is clearly very compressed near the far end of the transformed frustum (cf. Figure 2.8). In fact, it is more scaled the closer to the origin the near plane lies. As already mentioned, this means that we should always try to push the near plane as far away as possible. Having the far plane as close as possible is also helpful, but to a lesser degree.

In cases where the near plane is too close to the origin, we see a phenomenon known as *depth fighting* or *Z fighting*. It means that the depth buffer can no longer resolve which pixels come from objects that are occluded and which come from objects that are visible. The resulting images often look like holes have been cut in the objects that are supposed to be closer to the viewer. These artefacts are illustrated in Figure 2.14.

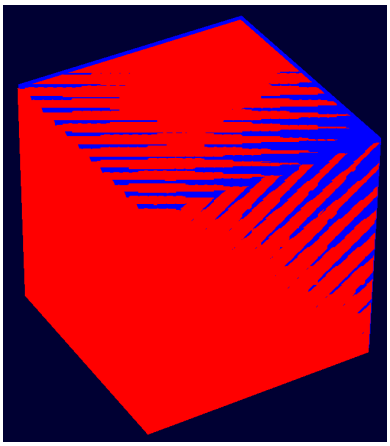


Figure 2.14: If the near plane is very close to the origin, the depth buffer precision near the rear end of the view volume becomes very poor. The result is depth fighting artefacts. In this case the blue cube is behind the red cube but shows through in several places.

2.4 Per Vertex Shading

So far, we have not discussed how to compute the colors that finally get written to the frame buffer. However, we have discussed interpolation, and the traditional way of shading a pixel is to first compute the color per vertex and then interpolate this color to the individual pixels. This is the method we describe in the following, and it was the only way of shading before programmable graphics hardware. With the advent of programmable shading, per pixel shading is often the best solution because it gives more accurate results. However, the difference in implementation is slight, since in per pixel shading we interpolate positions and vectors before computing lighting, and in per vertex lighting, we interpolate the computed color.

In either case, in real-time graphics, only local illumination from a point light source is usually taken into account. Local means that objects do not cast shadows, and light does not reflect between surfaces, i.e. illumination comes only from the (point) light source and is not influenced by other geometry than the vertex at which we want to compute the shaded color.

We compute the color of a vertex in eye coordinates. This choice is not arbitrary. Remember that we transform vertices directly from object to eye coordinates with the modelview matrix. Moreover, the shading depends on the relative position of the camera and the vertex. If we had chosen object coordinates instead, we would have to transform the light position and direction back into object coordinates. Since the modelview transform changes frequently, we would have to do this per vertex.

Instead, we now have to transform the *surface normal*, $\vec{n}_o$, into eye coordinates ($\vec{n}_e$). The normal is a 3D vector of unit length that is perpendicular to the surface and specified per vertex. Recall that perpendicular means that for any other vector $\vec{v}_o$ in the tangent plane of the point, we have that

$$\vec{v}_o \cdot \vec{n}_o = \vec{v}_o^T \vec{n}_o = 0$$

A bit of care must be taken when transforming the normal into eye coordinates. Since it is a vector and not a point, we need to set $w = 0$: this means that the vector represents a 3D vector as opposed to a point. Conveniently, if we simply set $w = 0$ in the homogeneous representation of the normal, and multiply with the modelview matrix

$$\vec{n}_e = \vec{MV}\vec{n}_o = \vec{MV} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

we get just the rotation and scaling components of the transformation and not the translation part. Unfortunately, this only works if the transformation does not contain anisotropic scaling. If we scale the Y axis more than the X axis, for instance to transform a sphere into an ellipsoid, the normals are no longer perpendicular to the surface.

Assume we have a pair of points $\vec{p}_o^1$ and $\vec{p}_o^2$ in object coordinates. If the vector from $\vec{p}_o^1$ to $\vec{p}_o^2$ is perpendicular to the normal, we can express this relationship as follows

$$(\vec{p}_o^1 - \vec{p}_o^2)^T \vec{n}_o = 0 \ .$$

Let us say we obtained these points by inverse transformation from eye space with the inverse modelview matrix:

$$((\vec{MV})^{-1}\vec{p}_e^1 - (\vec{MV})^{-1}\vec{p}_e^2)^T \vec{n}_o = 0$$

which is the same as

$$(\vec{p}_e^1 - \vec{p}_e^2)^T ((\vec{MV})^{-1})^T \vec{n}_o = 0$$

Thus, we can transform the normal by, $((\vec{MV})^{-1})^T$, the transpose of the inverse. This is guaranteed to work if the modelview matrix is non-singular (i.e. has an inverse) which it is guaranteed to be if we just rotate, translate and scale. As a final step the tranformed normal is renormalized by dividing it with its own length. This step can be omitted if we know that the modelview transform does not include scaling.

As mentioned, we compute shading using a point light source. Thus, another thing that needs to be transformed is the light source position which we also need in eye coordinates. It is often a source of confusion how one specifies a light source that is stationary with respect to the scene or with respect to the camera. In fact, the rules are simple, and we can break it down into three cases:

- If we specify the light source position directly in eye coordinates, then the light clearly does not move relative to the camera – no matter what model and view transformations we apply to the scene.

- If we specify the light source in world coordinates, i.e. applying just the view transformation, the result is a light source that is fixed relative to the scene in world coordinates.

- If we want a dynamically moving light source, we can add further modelling transformations which move the light relative to the scene.

2.4.1 *Phong Illumination Model*

Now we know all the things we need in order to compute the illumination at a vertex. The following are all 3D vectors (we forget the w coordinate).

- The eye (or camera) position is the origin $[0\ 0\ 0]$.
- The vertex position is $\vec{p}_e = \vec{MV}\vec{p}_o$.
- The normal $\vec{n}_e = ((\vec{MV})^{-1})^T\vec{n}_o$.
- The light source position $\vec{pl}_e$.

To simplify things in the following, we drop the e subscript which indicates that points or vectors are in eye space coordinates. Also, we will not need homogeneous coordinates, so vectors are just 3D vectors in the following.

From the position of the vertex, we can easily compute the normalized view vector pointing towards the eye

$$\vec{v} = -\frac{\vec{p}}{\|\vec{p}\|} \quad . \tag{2.12}$$

From the light source position we can compute the normalized direction towards the light source

$$\vec{l} = \frac{\vec{pl} - \vec{p}}{\|\vec{pl} - \vec{p}\|} \quad . \tag{2.13}$$

The vectors involved in lighting computation are shown in Figure 2.15.

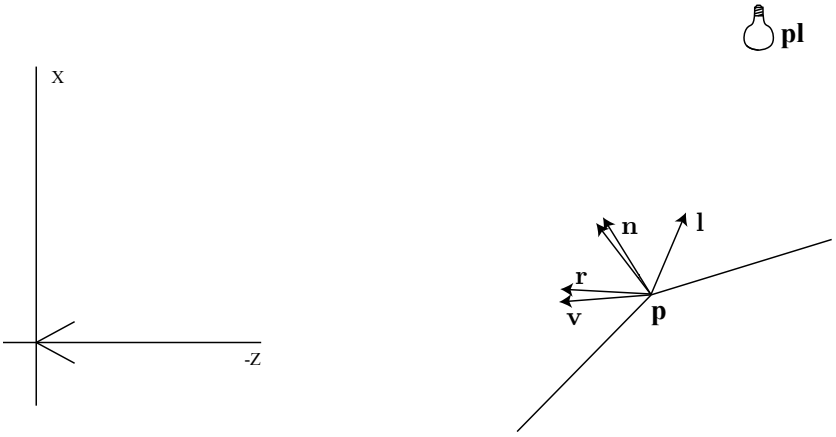


Figure 2.15: The vectors needed to compute shading according to the Phong and Blinn-Phong illumination model.

The simplest contribution to the illumination is the ambient light. The amount of "reflected" ambient light is

$$L_a = k_a I_a \ ,$$

where I_a is the intensity of ambient light in the environment and k_a is a coefficient which controls how much ambient light is reflected. Ambient light is an extremely crude approximation to *global illumination*. Global illumination is the general term used for light that is reflected by other surfaces before reaching the point from which it is reflected into the eye. For instance, if we let the light reaching the walls of a room illuminate the floor, we take global illumination into account. Global illumination is generally very expensive to compute and in real-time graphics, we mostly use crude approximations. The ambient term is the crudest possible such approximation, and it is a bit misleading to say that ambient light is actually reflected. But, without ambient light and reflection of ambient light, a surface will be completely dark unless illuminated by a light source which is often not what we want. However, from a physical point of view, ambient light is so crude that it does not really qualify as "a model".

The contribution from diffuse reflection is somewhat more physically based. An ideal diffuse surface reflects light equally in all directions; the intensity of light we perceive does not depend on our position relative to the point of reflection. On the other hand the amount of reflected light does depend on the angle, θ, between the light direction and the surface normal. If we denote the amount of diffusely reflected light L_d, then

$$L_d = k_d \cos(\theta) I_d = k_d (\vec{n} \cdot \vec{l}) I_d \ ,$$

where k_d is the diffuse reflectance of the material and I_d is the intensity of the light source. The diffuse reflection gradually decreases as we tilt the surface away from the light source. When the light source direction is perpendicular to the normal, the contribution is zero.

Surfaces are generally not just diffuse but also have some specular component. Unlike a diffuse reflection where light goes equally in all direction, a specular reflection reflects light in approximately just one direction.

This direction, $\vec{r}$, is the direction *toward* the light source reflected in the plane perpendicular to the normal, $\vec{n}$,

$$\vec{r} = 2(\vec{l} \cdot \vec{n})\vec{n} - \vec{l} \ .$$

The specular contribution is

$$L_s = k_s (\vec{r} \cdot \vec{v})^p I_s \ ,$$

where p is the Phong exponent or shininess. If this exponent is large, the specular reflection tends to be very sharp. If it is small it is more diffuse. Thus, p can be interpreted as a measure of how glossy or perfectly specular the material is. Another interpretation is that it provides a cue about the size of the light source. I_s is the light intensity that is subject to specular reflection. Of course, in the real world we do not have separate specular and diffuse intensities for a light source but this gives added flexibility.

It is important to note that there is a different way (due to Blinn) of computing the specular contribution. The half angle vector $\vec{h}$ is defined as the normalized average of the view and light vectors

$$\vec{h} = \frac{\vec{v} + \vec{l}}{\|\vec{v} + \vec{l}\|}$$

using the half angle vector, we get this, alternative, definition of the specular contribution:

$$L_s = k_s(\vec{h} \cdot \vec{n})^p I_s \ .$$

One advantage of this formulation is that the half angle vector is often constant: in many cases, we assume that the direction towards the viewer is constant (for the purpose of lighting only) and that the direction towards the light source is also constant (if the light is simulated sunlight this is a sound approximation). In this case, $\vec{v}$ and $\vec{l}$ are both constant, and as a consequence so is $\vec{h}$ and in many cases this gives a performance improvement. Another reason to prefer Blinn-Phong is that the $(\vec{h} \cdot \vec{n})^p$ term can be seen as the fraction of microfacets that point in the halfvector direction. Thus, if we assume a micro-facetted material, Blinn-Phong is a more correct approximation.

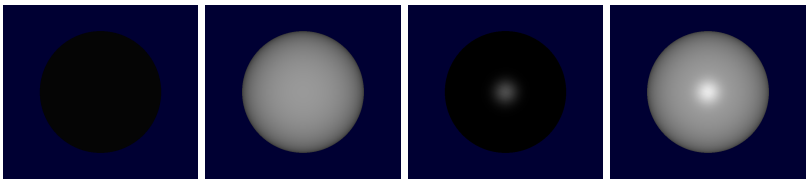


Figure 2.16: From left to right: The contributions from ambient (almost invisible), diffuse, and specular reflection. On the far right the combination of these lighting contributions.

If we combine the specular, diffuse, and ambient contributions we get the following equation for computing the color at a vertex

$$L = L_a + L_d + L_s = k_a I_a + k_d(\vec{n} \cdot \vec{l}) I_d + k_s(\vec{h} \cdot \vec{n})^p I_s \qquad (2.14)$$

Figure 2.16 illustrates these terms and their sum. Of course, if we only use scalars for illumination the result is going to be very gray. Instead of scalar coefficients k_a, k_d, and k_s we can use RGB (red,

green, blue) vectors which represent the ambient diffuse and specular colors of the material. Likewise, I_a, I_d, and I_s are RGB vectors containing the color of the light.

Note also that the simple model which underlies (2.14) is not energy preserving. We still need to multiply by normalization constants to ensure that the diffuse and specular terms integrate to unity [3]. This is a topic which we will return to in a later chapter.

[3] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 2008

2.5 *Texture Mapping*

Having computed a color per vertex using the Phong illumination, we could simply shade our pixels by interpolating this color in the way previously described. In many cases, we would like to add a bit more detail, though. Figure 2.17 shows the effect of adding texture. There is a dramatic difference between a smooth surface which has been shaded and the same surface with texture added. For this reason, texture mapping has been a standard feature of graphics hardware since the beginning.

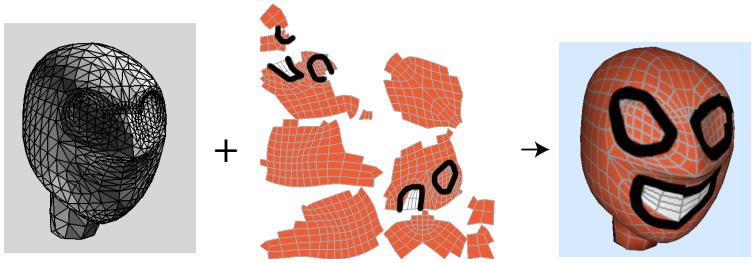


Figure 2.17: This figure illustrates the principle of texture mapping. Texture stored in a planar image is mapped onto the mesh.

In normal parlance the word texture refers to the tactile qualities of an object, but in the context of computer graphics, texture has a particular meaning which is the only meaning used below. In CG, textures are simply images which we map onto 3D models.

The principle behind the mapping is simple: Just like a vertex has a geometric position in 3D object space, it also has a position in texture space indicated via its *texture coordinates*. Texture coordinates are usually 2D coordinates in the range $[0,1] \times [0,1]$ or recently often $[0,W] \times [0,H]$ where W and H refer to the width and height of the texture image.

When a triangle is rasterized, these texture coordinates are interpolated along with the other attributes such as the shaded color computed for the vertices. We then look up the texture color in the texture image as illustrated in Figure 2.18.

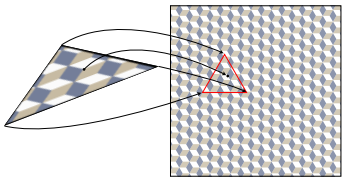


Figure 2.18: Texture coordinates provide a mapping from the geometric position of a vertex to its position in texture space. When a triangle is drawn, we can interpolate its texture coordinates to a given pixel and hence find the corresponding position in the texture image.

It is very important that the texture coordinates are interpolated in a perspective correct way. Otherwise, we get very peculiar images like the one shown in Figure 2.19.

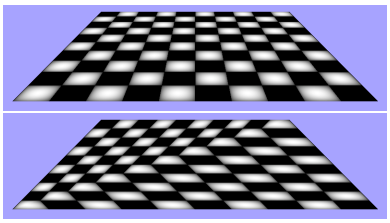


Figure 2.19: Perspective correct interpolation is important for texture coordinates. On the right we see what happens if the texture coordinates are interpolated linearly in window space.

Once we have looked up a texture color, we can use it in a variety of ways. The simplest is to simply set the pixel color to the texture color. This is often used in conjunction with alpha testing to do billboarding. A billboard is simply an image which we use to represent an object. Instead of drawing the object, we draw an image of the object as illustrated in Figure 13.2. For an object which is far away, this is sometimes acceptable, but it is necessary to mask out those pixels which correspond to background. This masking is done by including an alpha channel in the texture image where alpha is set to 0 for background pixels and 1 for foreground pixels. Alpha testing is the used to remove pixels with value 0 since we can filter pixels based on their alpha value in the fragment shader and thus cut out the background parts of a texture image as illustrated in Figure 13.2.

The typical way of using the texture color, however, is to multiply the shading color with the texture color. This corresponds to storing the color of the material in the texture, and it is this mode that is used in Figure 2.17.

We can do many other things with texture - especially with the advent of programmable shading (cf. Section 2.6).

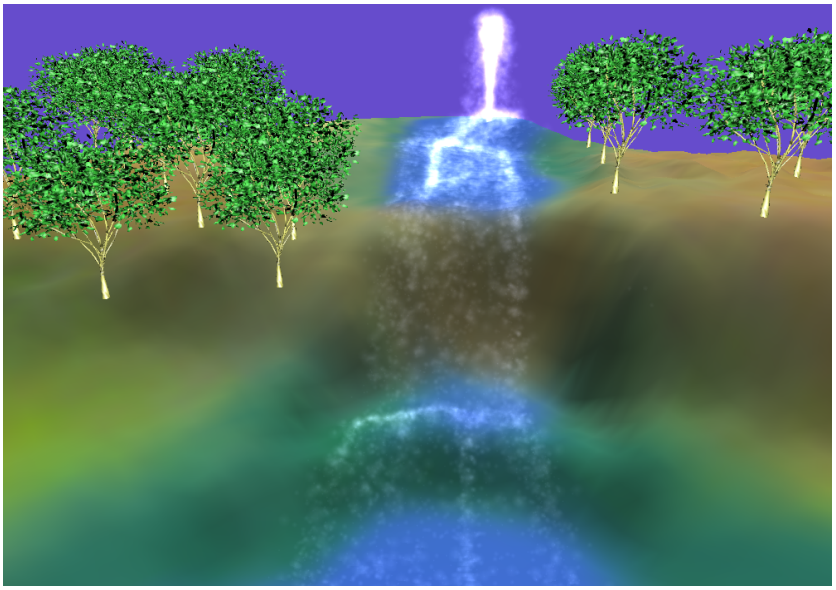


Figure 2.20: There are no 3D models of trees in this image. Instead a single tree was drawn in an image which is used as billboard. This way of drawing trees is a bit dated. Nowadays one would use many more polygons to define the tree.

2.5.1 *Interpolation in Texture Images*

Of course, the interpolated texture coordinates usually lie somewhere between the pixels in the texture image. Consequently, we need some sort of interpolation in the image texture.

The simplest interpolation regards a *texel* (pixel in texture image) as a small square, and we simply pick the pixel color corresponding to what square the sample point lies in. If we regard the texture image as a grid of points where each point is the center of a texel, this is nearest neighbor interpolation. Unfortunately, the texture image is usually made either bigger or smaller when it is mapped onto the 3D geometry and then projected onto the screen. If the texture image is magnified, the result will be a rather blocky image.

This can be fixed through interpolation. GPUs invariably use bilinear interpolation[4] or a method based on bilinear interpolation. Bilinear interpolation is a simple way of interpolating between values at the corners of a square to a point inside the square. It is really a composition of three linear interpolations:

$$f = (1-\beta)((1-\alpha)f_0 + \alpha f_1) + \beta((1-\alpha)f_2 + \alpha f_3) \tag{2.15}$$

where f_i are the quantities we interpolate and the weights are α and β. See Figure 5.3 for an illustration.

[4] Despite linear being part of the name, bilinear interpolation is really quadratic. This need not detain us, however.

If the texture image is magnified, bilinear interpolation is about the best we can do. However, if the texture is *minified*, i.e. made smaller, both nearest neighbor and bilinear interpolation give very poor results. This is illustrated in the two leftmost images of Figure 2.22. The problem is really *aliasing* - high frequencies in the texture image which lead to spurious low frequency details in the rendered image. Informally, when a texture is made very small, we skip texels when the image is generated and this can lead to the strange patterns shown in the figure.

The solution is to use a smaller texture which is blurred (or low pass filtered) before it is subsampled to a resolution where the texels are of approximately the same size as the pixels.

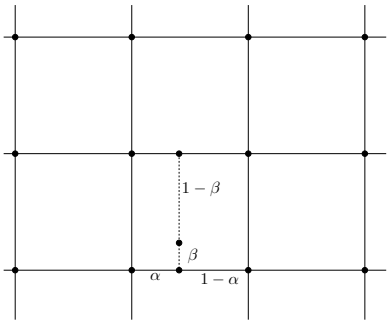


Figure 2.21: In bilinear interpolation, we interpolate between four data points which lie on a regular grid. It is implemented in terms of three linear interpolations. We first interpolate to two intermediate points and then between these two intermediate points.

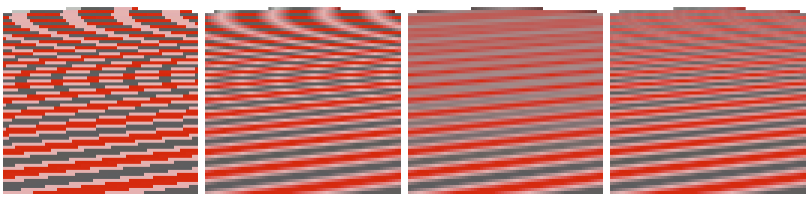


Figure 2.22: From left to right: Nearest texel, bilinear interpolation, mipmapping, and anisotropic interpolation.

In practice, we cannot compute this right-sized texture on the fly.

Instead, graphics hardware precomputes a pyramid of textures. The bottom (zero) level is the original texture. Level one is half the size in both width and height. Thus one texel in level one covers precisely four texels in level zero, and the level one pixel is simply set to the average color of these four texels. If we do this iteratively, the result is a pyramid of textures ranging from the original texture to one with a single texel. Of course, this requires the original texture to have both width and height which are powers of two. However, it is not important that the images are square. For instance if the image has power-of-two dimensions but is twice as broad as high, say 128×64, we end up with two pixels instead of one in the highest level but one and then average these two pixels to get the topmost level. Arbitrary size textures are rescaled before mipmap computation.

When using mipmaps, we first find the place in the texture where we need to do a look up and also the approximate size of a pixel at that point projected into texture space. Based on the size, we choose the two levels in the mipmap whose texels are closest to the size of a projected pixel (one above and one below), interpolate separately in each image, and then interpolate between these two levels in order to produce the final interpolation. In other words, we perform two bilinear interpolations in separate mipmap levels followed by an interpolation between levels for a total of seven interpolations involving eight texels. This is called trilinear interpolation.

Anisotropic Texture Interpolation

Mipmapping is a very important technique, but it is not perfect. As we see in Figure 2.22 mipmapping avoids the nasty artefacts of linear interpolation very effectively, but it also introduces some blurring. The problem is that we rarely compress an image evenly in both directions.

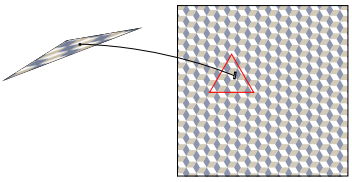


Figure 2.23: A single square pixel mapped back into texture space becomes a very stretched quadrilateral (four sided polygon).

In Figure 2.19 left, we see the perspective image of a square divided into smaller squares. Clearly these are more compressed in the screen Y direction than the X direction. Another way of saying the same thing is that the pixel footprint in texture space is more stretched in the direction corresponding to the screen Y direction. There is another illustration of the issue in Figure 2.23. A square pixel inside the triangle corresponds to a long rectangle in texture space. Mipmapping does not solve the problem here because it scales the image down equally in width and height. Put differently, if we use just one mipmap level when taking a pixel sample, we will choose a level which is too coarse because the pixel appears to have a big footprint in texture space, but we do not take into account that

the large footprint is very stretched. The solution is to find the direction in which the pixel footprint is stretched in texture space and then take several samples along that direction. These samples are then averaged to produce the final value.

Effectively, this breakes the footprint of a pixel in texture space up into smaller bits which are more square. These smaller bits can be interpolated at more detailed levels of the mipmap pyramid. In other words, we get a sharper interpolation using anisotropic texture mapping because we blur more in the direction that the texture is actually compressed. An example of the result of anisotropic texture interpolation is shown in Figure 2.22 far right.

Clearly, anisotropic texture interpolation requires a big number of samples. Often, we take 2, 4, 8, or 16 samples in the direction that the texture is compressed. Each of these samples then use eight pixels from the mipmap. However, as of writing, high end graphics cards are up to the task of running highly realistic video games using anisotropic texture interpolation at full frame rate.

2.6 *Programmable Shading*

One of the first graphics cards was the Voodoo 1 from 3dfx Interactive, a company that was later bought by NVIDIA. The Vodoo 1 did not perform many of the tasks performed by modern graphics cards. For instance it did not do the vertex processing but only triangle rasterization and still required a 2D graphics card to be installed in the computer.

Contemporary graphics cards require no such thing and in fact all the major operating systems (Apple's Mac OS X, Windows Vista, and Linux in some guises) are able to use the graphics card to accelerate aspects of the graphical user interface.

However, the graphics cards have not only improved in raw power they have improved even more in flexibility. The current pipeline is completely programmable, and the fixed function pipeline which is largely what we have described till now is really just one possible *shader* to run. In the following, we will use the word shader to denote a small program which runs on the GPU. There are two main types of shaders which we can run

- Vertex shaders, which compute both the transformation of vertices and the computation of illumination as described in this text.
- Fragment shaders, which compute the color of a fragment often by combining interpolated vertex colors with textures.

These programs run directly on the graphics cards and they are written in high level programming languages designed for GPUs rather

than CPUs such as the OpenGL shading language (GLSL), High Level Shading Language (HLSL), or C for Graphics (CG). The first of these GLSL is OpenGL specific. HLSL and CG are very similar, but the former is directed only at DirectX and the latter can be used with both OpenGL and DirectX.

Other types of programs besides vertex and fragment programs have emerged. Geometry shaders run right after vertex shaders and for a given primitive (in general a triangle) the geometry shader has access to all the vertices of the triangle. This allows us to perform computations which are not possible if we can see only a single vertex as is the case with vertex shaders. In particular, we can subdivide the triangles into finer triangles - amplifying the geometry. More recently, shaders which allow us to directly tessellate smooth surface patches into triangles have become available.

The entire pipeline is now capable of floating point computations. This is true also in the fragment part of the pipeline, which allows us to deal with high dynamic range colors. This is extremely important since fixed point with eight bit gives us a very narrow range of intensities to work with. If the light source is the sun it is less than satisfying to only have 256 intensity levels between the brightest light and pitch black. With 16 or even 32 bit floating point colors, we can do far more realistic color computations even if the final output to the frame buffer is restricted to eight bit fixed point per color channel due to the limitations of most monitors.

It is also important to note that even if we have to convert to eight bit (fixed point) per color channel for output to a *screen displayed* framebuffer, we do not have such a restriction if the output is to a framebuffer not displayed on the screen. If we render to an off-screen framebuffer, we can use 32 bit floating point per color channel – assuming our graphics card supports it.

This is just one important aspect of off-screen framebuffers. In fact, the ability to render to an off-screen framebuffer is enormously important to modern computer graphics, since it has numerous applications ranging from non-photorealistic rendering to shadow rendering. The reason why it is so important is that such an off-screen framebuffer can be used as a texture in the next rendering pass. Thus, we can render something to an off-screen framebuffer and then use that in a second pass. Many advanced real-time graphics effects require at least a couple of passes using the output from one pass in the next.

2.6.1 *Vertex and Fragment Shaders*

The input to a vertex program is the vertex *attributes*. Attributes change per vertex and are thus passed as arguments to the vertex program. Typical attributes are position, normal, and texture coordinates. However, vertex programs also have access to other variables called *uniforms*. Uniforms do not change per vertex and are therefore not passed as attributes. The modelview and projection matrices, as well as material colors for shading are almost always stored as uniforms. The vertex program can also perform texture lookup although this is not used in a typical pipeline. The mandatory output from a vertex program is the transformed position of the vertex. Typically, the program additionally outputs the vertex color and texture coordinates.

The vertices produced as output from the vertex shader (or geometry shader if used) are assembled into triangles, and these triangles are then clipped and rasterized, and for each pixel, we interpolate the attributes from the vertices. The interpolated attributes form the input to the fragment shader. The fragment shader will often look up the texture color based on the interpolated texture coordinates and combine this color with the color interpolated from the vertices. The output from the fragment shader must be a color, but it is also possible to output a depth value and other pixel attributes. Recent years have seen the introduction of multiple rendering targets which allow us to write different colors to each render target. Since we can only have one visible framebuffer, multiple render targets are mostly of interest if we render to off-screen framebuffers.

2.6.2 *Animation*

Perhaps the first application of vertex shaders was animation. An often encountered bottleneck in computer graphics is the transfer of data from the motherboard memory to the graphics card via the PCI express bus. Graphics cards can cache the triangles in graphics card memory, but if we animate the model, the vertices change in each frame. However, with a vertex shader, we can recompute the positions of the vertices in each frame.

A very simple way of doing this is to have multiple positions for each vertex. When drawing the object, we simply interpolate (linearly) between two vertex positions in the shader. This provides a smooth transition.

Another common technique for GPU based animation is skeleton-based animation. What this means is that we associate a skeletal structure with the mesh. Each vertex is then influenced by several bones of the skeleton. We store (as uniforms) a transformation matrix

for each bone and then compute an average transformation matrix for each vertex where the average is taken over all the matrices whose corresponding bones affect that vertex.

2.6.3 Per pixel lighting

It is highly efficient to compute lighting per vertex, but it also introduces some artefacts. For instance, we only see highlights when the direction of reflected light is directly towards the eye. This could happen at the interior of a triangle. However, we compute illumination at the vertices, and if the highlight is not present at the vertices the interpolated color will not contain the highlight even though we should see it.

The solution is to compute per pixel lighting. To do so, we need to interpolate the normal rather than the color to each pixel and then compute the lighting per pixel. This usually produces far superior results at the expense of some additional computation.

2.6.4 Deferred Shading and Image Processing

As mentioned, it is enormously important that we can output to an off-screen framebuffer. One application of this feature is that we can output an image containing data needed for shading and do the shading in a second pass. For instance, we can output the position of the fragment - i.e. the interpolated vertex position - and the vertex normal. With this data, we can compute shading in a second pass. In the second pass, we would typically just draw one big rectangle covering the screen and for each pixel, we would look up the position and normal in the texture produced by rendering to an off-screen framebuffer in the first pass.

At first that might seem to simply add complication. However, note that not all fragments drawn in the first pass may be visible. Some will be overwritten by closer fragments that are later drawn to the same pixel. This means that we avoid some (per pixel) shading computations that way. Moreover, the pixel shader in the initial pass is very simple. It just outputs geometry information per pixel,. In the second pass, the geometry is just a single rectangle. Consequently, all resources are used on fragment shading. It seems that this leads to greater efficiency - at least in modern graphics cards where load balancing takes place because the same computational units are used for vertex and fragment shading.

Moreover, we can use image processing techniques to compute effects which are not possible in a single pass. A good example is edge detection. We can compute, per pixel, the value of an edge detection filter on the depth buffer but also on the normal buffer. If a

discontinuity is detected, we output black. This can be used to give our rendering a toon style appearance, especially if we also compute the color in a toon-style fashion as shown in Figure 2.24.

Figure 2.24: A dragon rendered in two passes where the first pass outputs vertex position and normal to each pixel. The second pass computes a toon style shading and the result of an edge detection filter on both the per pixel position and per pixel normals. The result is a toon shaded image where sharp creases and depth discontinuities are drawn in black.

2.7 *Efficent Rendering*

So far, we have only discussed how to render a single triangle. However, a modern graphics card is able to render hundreds of millions of triangles per second (and output billions of pixels). To actually get these numbers however, we have to be sensible about how we send the data to the graphics card.

A pertinent observation in this regard is that most vertices are shared by several triangles. A good rule of thumb is that six triangles generally share a vertex. In most cases, we want to use the exact same vertex attributes for each of these six triangles. For this reason, there is a cache (Sometimes called the transform and lighting (T&L) cache but now, more reasonably, the post-vertex-shader cache. C.f. Figure 2.1)

To exploit this cache, however, we must be able to signal that the vertex we need is one that was previously processed by vertex shading. There are two ways in which we can do this: Using triangle

strips or indexed primitives.

2.7.1 *Triangle Strips*

Figure 6.2 shows a triangle strip. To use triangle strips, we first need to inform the graphics card that the *geometric primitive* we want to draw is not a triangle but a triangle strip. Next, we send a stream of vertices, in the example from the figure, we send the vertices labeled 0,1,2,3,4,5, and 6. In this case, the triangles produced are 012, 213, 234, 435, 456. In other words, the graphics hardware always connects the current vertex with the edge formed by the past two vertices, and the orientation is consistent.

Every time a triangle is drawn, the GPU only needs to shade one new vertex. The other two are taken from cache.

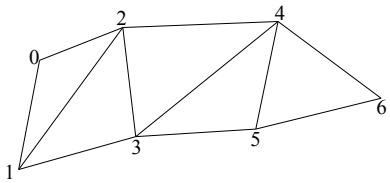


Figure 2.25: A triangle strip is a sequence of vertices where every new vertex (after the first two vertices) gives rise to a triangle formed by itself and the two preceding vertices.

2.7.2 *Indexed Primitives*

Computing long strips of triangles that cover a model is not a computationally easy task. Nor is it, perhaps, so important. Another way in which we can exploit the cache is to use vertex arrays. In other words, we send an array of vertices to the graphics card and then an array of triangles. However, instead of specifying the geometric position of each vertex, we specify an index into the array of vertices.

This scheme can be used both with and without triangle strips (i.e. a strip can also be defined in terms of indices). In either case, locality is essential to exploiting the cache well. This is not different from any other scenario involving a cache. The longer we wait before reusing a vertex, the more likely that it has been purged from the cache. Of course, optimal use of the cache also means that we should be aware of what size the cache is.

2.7.3 *Retained Mode: Display Lists, Vertex Buffers, and Instances*

Efficient rendering does not only require our geometry information to be structured well as we have just discussed. It also requires communication between the main memory and the graphics card to be efficient.

It is a bit old school to talk about *immediate* and *retained* mode, but these two opposite notions are still relevant to understanding how to achieve efficient rendering.

In immediate mode, the triangles sent to the graphics card are immediately drawn, hence the name. This can be extremely convenient because it is easier for a programmer to specify each vertex with a function call than to first assemble a list of vertices in memory and

a list of triangles in memory and then communicate all of that to the graphics card. Unfortunately, immediate mode is slow. A function call on the CPU side per vertex is simply too costly. Moreover, sending the geometry every frame is also not a tenable proposition. For this reason, and for all its convenience, immediate mode is not a feature in Microsofts Direct3D API, OpenGL for embedded systems, or even recent versions of the core profile of the normal OpenGL API.

There is an easy fix to the performance problem of immediate mode, namely display lists. A display list is essentially a macro which you can record. Through a function call, you instruct the graphics API (only OpenGL in this case) that you want to record a display list. All subsequent graphics commands are then recorded for later playback and nothing is drawn. Another function call stops the recording. Finally, you can replay the display list with yet another function call. This is very simple for the programmer, and display lists combined with immediate mode is a powerful tool for efficient rendering since the display lists are almost always cached in graphics card memory.

Unfortunately, a facility which allows us to record general graphics commands for later playback appears to be somewhat difficult to implement in the graphics driver. Probably, this is the reason why display lists are now deprecated and removed from recent version of the OpenGL core library. Above, we briefly mentioned arrays of vertices and triangles. That is now the tools used for efficient rendering. We need to store these arrays on the graphics card for the best performance. For this reason, all modern graphics APIs supply functions which allow you to fill buffers which are subsequently transferred to graphics card memory.

However, this only provides a facility for drawing a single copy of some geometric object efficiently. Say I fill a buffer with a geometric model of a car, and I want to draw many instances of that car in different position and different colors. This is where the notion of *instancing* comes in. Instancing, which is also supported by all modern graphics APIs, allows you to render many instances of each object in one draw call. Each object can have different parameters (e.g. transformation matrix, material colors etc.) and these parameters are stored in a separate stream. Essentially, what instancing does is re-render the geometry for each element in the parameter stream.

2.8 Aliasing and Anti-Aliasing

Rendering can be seen as sampling a 2D function. We have a continuous function in a bounded 2D spatial domain, and we sample this function at a discrete set of locations, namely the pixel centers.

When sampling, it is always a problem that the function might contain higher frequencies than half of our sampling frequency, which, according to the Nyquist of theorem, is the highest frequency that we can reconstruct.

While the Nyquist theorem sounds advanced, it has a very simple explanation. Continuous periodic functions which map a point in a 1D space (the line of real numbers) to real numbers can be expressed as infinite sums of sine (and cosine) functions at increasing frequency. This known as the Fourier series of the function. Now, for a given frequency of a sine function, if we have two samples per period, we know the frequency of the function. Consequently, if the Fourier series does not contain higher frequencies than half the sampling frequency, we have two samples per period for every single sine function in the series, and we can reconstruct the true continuous function from its Fourier series. See Chapter 6 of [5].

The sampling theorem generalizes to 2D (where we have 2D analogs of sine functions) and explains why we would like to have images which are limited in frequency. That is not possible in general, because the discontinuity in intensity between the triangle and the background is a feature in the image which is unbounded in the frequency domain, and when it is sampled and reconstructed we get artefacts - so called jaggies or staircase artefacts where the triangle and background meet. This is illustrated in Figure 2.26.

Mipmapping is our solution for texture, but it works only for texture. If we were able to low pass filter the edge producing a smoother transition before sampling, the edge would look much better. Unfortunately, this is not possible in any practical way. However, what we can do is to draw the triangle at a much higher resolution (say we draw an image twice as wide and twice as high as needed) and then average groups of four pixels to produce a single average pixel. This is known as *supersampling*. Supersampling would clearly produce a fuzzy gray value instead of sharp discontinuities. It does not fix the problem, but it moves the problem to higher frequencies where it is less visible.

Unfortunately, four samples per pixel are often not enough in spite of the fact that it is much more expensive to compute.

Modern graphics hardware can use (often) up to sixteen samples per pixel. This leads to much better results. Also, there are smarter ways of sampling than just producing images at higher resolution. A crucial observation is that we only need the additional samples near edges and that we only need to sample geometry at super resolution since mipmapping (and anisotropic interpolation) takes care of tex-

[5] J. M. Carstensen. *Image analysis, vision, and computer graphics*. Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2001

Figure 2.26: The difference between no anti-aliasing on the top and anti-aliasing with 16 samples per pixel on the bottom.

tures. These observations are what led to *multisampling* which is the term for a family of supersampling methods that only take the geometry into account and only near edges. When multisampling only a single texture sample is generally used, and the fragment program is only run once.

2.9 *Conclusions*

This brief lecture note has only scraped the surface of real-time computer graphics. Hopefully, this is still sufficient to give you an overview of the basic principles and possibilities. For more details, we refer you to the references below.

3
Getting Started

This chapter gives an introduction on how to get started with OpenGL using the GLFW framework.

Since this book provides a framework for OpenGL using GLFW, the examples in this chapter are based on that framework, and we will go through the libraries which the framework contains or employs to ensure that you are ready to start working. The most important part is how the framework connects CPU bound code to GLSL shader programs. Having read this chapter and completed the exercise, you should also be able to work with shader programs. This is important since the framework allows you to focus on the algorithms for real-time graphics which are - to a large extent - implemented in shaders. On the other hand, this chapter does not contain many details regarding how to send geometry down the graphics pipeline. That follows in Chapter 5 where we discuss efficient rendering of geometry.

Unfortunately, while our framework is essential in order to reach interesting graphics challenges, it also does hides some subtle details that you would be exposed to if starting from scratch. We will do our best to help expose these details along the way, but we also encourage the reader to review the introductory chapter "The Real-Time Graphics Pipeline" where many of the nitty gritty details of real-time graphics are discussed. With an understanding of these, you should be on a sound footing and poised to learn the techniques in this book.

3.1 Getting Started on the Desktop

OpenGL is a great API which allows you to create amazing graphics with tons of beautiful special effects all rendered in realtime. But getting started with OpenGL programming can be challenging since even the simplest OpenGL program are easily more than a 100 lines of code. There are a couple of different reasons for this:

- OpenGL is a graphics API that let you send commands to the graphics card through the driver. However OpenGL does not have any way create an OpenGL context, such as a window or a full-screen application. To create an OpenGL context you need to use some platform specific APIs (such as Win32 for Windows, GLX for Linux and Cocoa for OSX) or use a window toolkit (GLUT, Qt, GLFW or SDL) for managing this task. All approaches require that you learn some new APIs and learn how to compile and link your code with these libraries.
- Over the years OpenGL has become a much more low level API, which makes it much flexible and makes the driver much more slim (and easier to optimize for high performance). The problem with using a low level API is that it requires much more code in a number of cases. Compared to modern OpenGL, the old school OpenGL
  - used a configurable fixed function pipeline to transform vertices into pixels. This approach had a number of API calls for managing light-properties, transformations, fog, etc. In modern OpenGL shaders have replaced the fixed function pipeline, which means that the programmer is now responsible for transformation of vertices as well as the lighting calculations. The shaders both needs to be compiled at runtime and configured for each draw call.
  - used to have a matrix stack used for vertex (and other) transformations. OpenGL had functions for managing this stack using common operations (such as translate, rotate, scale, perspective projection and orthographic projection). This has also been removed, which means that you either need to create a simple matrix library yourself or use a 3rd party library. In the provided framework, we use a library called CGLA which is also available as a part of GEL. A similar library called GLM is part of the so-called OpenGL SDK.
  - sent geometry data to the GPU one vertex at a time (except when using display lists). This is now replaced by data buffers on the GPU, which needs to be uploaded and configured to tell OpenGL how the data is organized.

To a large extent these changes mean that we can not just start writing interesting OpenGL programs from scratch, and that is why we provide a rather comprehensive framework that takes care of many things including model (and other asset) loading, transformations (including setting up the view transform and projection), sending the model to the graphics processor and loading shaders.

3.1.1 Window Toolkits

Windows toolkits are software libraries which simplifies development of programs with a graphical user interface. The following discussion is limited to windows toolkits with OpenGL support. Windows toolkits simplifies development in the following ways:

- **Abstraction:** Usually windows toolkits hides all the low level API details and instead provides a simpler and more cleaner API.
- **Event handling:** Windows toolkits also handles low level events such as input from mouse and keyboard as well as windows events (resize, close, etc). Appropriate event information can be queried or accessed using callback functions provided to the window toolkit.
- **OpenGL context creation:** Provides a way of creating an OpenGL context in either a windows or in fullscreen mode. A OpenGL context consists of a color framebuffer that contains the pixels to be draw on the screen. In addition a context can have the following options - the most important is listed below.
  - **Double buffer:** Allows OpenGL to work on one framebuffer while another framebuffer is being displayed. This approach solves problems with incomplete frames being displayed. When using double buffering the buffers are swapped after each frame.
  - **V-synch:** An option used to ensure that swapping framebuffers does not occur during a screen update (which results parts of both framebuffers being displayed - also called tearing).
  - **Depth buffer:** Used for ensure correct ordering of rendered geometry. Created with a given precision - usually 16, 24 or 36 bits.
  - **Stencil buffer:** Used for rendering effects where only parts of the framebuffer needs to be updated.
- **Platform independence:** Most windows toolkits have implementations on multiple platforms. This makes it easy to compile the program on different platforms with only few changes in source code.

There exists two types of window toolkits; libraries and frameworks.

Libraries provides a set of classes and functions that provides an abstraction over low level API functions and provides access to reusable functionality. An example of an library is SDL where the

programmer has to write his own main loop, which should poll the library for events, render the scene and wait for the next iteration to start.

Frameworks is in many ways the same as a library, but with one important distinction: The inversion of control. Frameworks will take control over the main thread and provide some callback mechanism when some events occur. This is called the Hollywood Principle: Don't call us, we'll call you. Using this terminology, GLUT which was one of the first window toolkits for OpenGL is a framework whereas we will be using GLFW which ironically is a library despite the fact that the name stands for GL FrameWork.

3.2 *Basics of GLFW*

Perhaps the simplest graphics program we can create is one which draws a window of uniform color. With GLFW and OpenGL we can achieve this modest goal as follows:

```
#include <GLFW/glfw3.h>
int main() {
    glfwInit();
    GLFWwindow* window = glfwCreateWindow(512, 512,
                                "simple", NULL, NULL);
    glfwMakeContextCurrent(window);
    glClearColor(0,1,0,0);
    while (!glfwWindowShouldClose(window)) {
        glClear(GL_COLOR_BUFFER_BIT);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    glfwTerminate();
}
```

To actually compile this code, we clearly need a C (or C++) compiler and GLFW must be installed. On some platforms, we can install GLFW as a standard package but on Windows, it seems that we have to decide where to place the include files and library files ourselves.

We can glean a few things from the program above. In order to use GLFW, we need to call the `glfwInit` function. Next, we have to create a window and make the context associated with that window current.

The while loop is where the drawing happens. Only in this case there is no drawing except the screen is cleared to the preset green color. However, we swap front buffer and back buffer and poll for "events" inside the loop. The swapping is because we generally draw

to the back buffer and only then swap to make it visible. The polling is in order to respond to events such as resizing of the window, click of a mouse button, a key press or movement of the mouse. In this case nothing happens because there are no events.

We can make the program more interesting by adding a function that handles events and registering it with the event handler. First, we add a function which handles cursor position update events:

```
#include <iostream>
using namespace std;
void cursor_pos(GLFWwindow* window,double x,double y) {
    cout << " X, Y : " << x << " " << y << endl;
}
```

Subsequently, we register this function.

```
glfwSetCursorPosCallback(window, cursor_pos);
```

This has to happen in `main` after the window has been created. Now, when we run the program, the position of the cursor will tirelessly be printed to the console whenever our green window has input focus and the mouse moves.

Unfortunately, OpenGL now comes in a number of flavors and it behooves us to be specific about which version of OpenGL we want to use. This is achieved via the following block of code:

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

These four function calls must be issued after `glfwInit` but before we create the window. Briefly put, these calls request that when a window is created, the OpenGL context associated with the window should be a version 4.1 context and that it be forward compatible: features that were deprecated in 4.1 should note be allowed. Also, we want a core profile which means that old school OpenGL calls are not allowed.

There are newer versions of OpenGL than 4.1, but not all hardware supports them. If you only want your program to run on a single machine that may not matter, but in general it seems prudent to ask for the oldest version that contains the needed features. However, it is not enough to get the right version of the context. We also need to make sure that the API (i.e. the OpenGL functions we can call) correspond to the context. Unfortunately, Windows only comes with an ancient version of the OpenGL API, and we need an extension

loader to bring the API up to date. It seems best to use the one called GLEW. The last step, then, is to include the GLEW header files and link against the GLEW library. We can then include the following snippet:

```
GLenum glewinit = glewInit();
if (glewinit != GLEW_OK) {
  cout << "Glew did not initialize: "
       << glewGetErrorString(glewinit) << endl;
  return -1;
}
else
  cout << "GLEW OK" << endl;
```

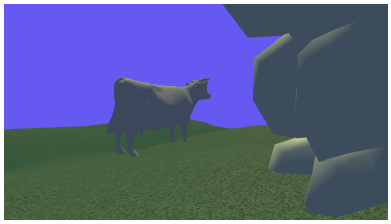
Note that in the example above, we have a bit of error handling. If GLEW did not initialize, the program prints an error message and bails out. We should have included two other error checks for GLFW. First of all if `glfwInit` fails, we should terminate the program and if window creation fails, we should also terminate. This might not seem so important, but if we omit it the program will just crash or (worse) not work. The advantage of these explicit error checks is that we can see precisely where the failure occurs.

3.3 *Exercise project: Desktop: Fragment Shading and Wireframe Rendering*

To solve the problems in this exercise, you need to find the solution for the IntroScene. If you are on the Windows platform it is time to start Visual Studio, and for Mac users this would be the time to open Xcode. On Windows go to the `Win/02564` directory where you will find `02564.sln`. On MacOS, the solution is in `MacOS` and called `02564.xcworkspace`. In either case, you will find IntroScene in the solution file.

Find the file called `IntrosScene.cpp`, compile the project and run the program. You can spin the model (of a cow) with the mouse and 'w' toggles between wireframe and normal rendering. Finally, 'r' reloads all shaders.

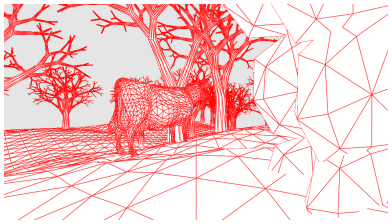
Part 1



The two shaders `object.vert` and `object.frag` are used to render the objects in the scene. However, the shading is per vertex. Change the program so that it shades per fragment instead.

Part 2

If you press 'w', the scene is drawn in wireframe with hidden surface removal, but there is something wrong with the wireframe GLSL program which is composed of the vertex, geometry, and fragment shaders `wire.vert`, `wire.geo`, and `wire.frag`. The principle is that if a fragment is close to a triangle edge, it is drawn in edge color. Otherwise, it is drawn white.



The distance from a fragment to the edges of a triangle can be computed by linearly interpolating the distances from the three vertices to each of the three edges of the triangle. In the geometry shader when computing the vector of distances for each vertex, two of the distances are set to zero (since the vertex is incident on two of the edges), but the third distance is set to 50. Change this value to the actual distance from the vertex to the opposite edge. Note that the distances must be computed in window space since we want the lines to have even width as seen on the screen.

Part 3

Try to answer the questions below as concisely as possible: If `M` is the modelview matrix, how is the normal matrix expressed in terms of `M`? What is the difference between whether there is `in`, `out`, or `uniform` in front of the type name in a variable declaration in the shader? Why is a geometry shader a good place to compute the distance in Part 2? What is a vertex attribute? Explain briefly the notions of immediate and retained mode. Explain briefly how a stateful API differs from a stateless API.

Deliverables

For parts 1 and 2, we need before and after screen shots that show the effects of your changes. Describe briefly what you did and how it changed the rendering. Note that before and after screen shots should be taken from the exact same location. For part 3, we need the answers. Submit the GLSL source code for the shaders and source code for scene composition if you complete Part 4. Deliver everything as a single PDF file. Note that this report should be considered a log - also as an aid to your own memory. Feedback is mostly given in the lab, so do not spend too much time writing prose.

4
Procedural Generation of Content

Very often, it is the content (assets) that make the difference between great looking and terrible computer graphics. A scene with carefully and professionally hand crafted models can often be extremely appealing. In research, such scenes are often hard to come by, and in compute games they are a very significant part of the cost.

However, hand crafted models and textures acquired from photographs are not only hard to come by, and time consuming to make, they also take up a lot of space which, in turn, increases load time. These are some of the reasons why procedurally generated content is appealing. If you can use an algorithm to create a texture, a 3D model, or an animation then you can use the algorithm to populate your application with vast numbers of slightly different models at little extra cost. Unfortunately, procedurally generated content often does not look quite as appealing as hand crafted content, but if we can use the procedures to speed up the manual work or to supplement it, then we are in a much better place than if everything has to be manually created. For these reasons, procedural content will probably have a bright future.

Procedural content is a very broad term, though. Essentially, if we use a computer program to create a 3D model or a texture, it is procedural content generation. However, we often use procedural generation for objects that are supposed to look like they are not man made, and, in this context, some concepts seem to be crucial.

- The first is noise. Computer graphics often looks too clean. Especially when it comes to natural phenomena. There is a certain variation and irregularity in all things that have come into existence through natural processes – as opposed to coming from a factory. In many cases we can use pseudorandom numbers to mimic this variation, which is what we discuss in the next section.
- The other important aspect is fractal structure which simply means that the same structures or variations of the same structures ap-

pear at different scales. So called *Perlin noise*, which he called *turbulence*, is really a multi scale noise function as will be explained shortly.

if we want to model things that have a sharply defined surface, we often need to use a different approach. Things that grow are often modeled using grammar based techniques, where the object is represented as a string of tokens that gets rewritten into a new string. For instance, the initial string might represent a trunk, and after a rewrite it contains the trunk and some branches; rewriting again might add twigs. Thus, because we often keep the old structure when rewriting, the grammar based techniques (for trees we almost always use L-Systems) also imbue the generated objects with a multi scale structure. Grammar based techniques are the main topic of the final section in this chapter.

4.1 Random Numbers

At the heart of noise is random number generation.[1] In computer graphics, we typically use what is known as pseudo random numbers. Such numbers are really periodic sequences of integers which are entirely deterministic, but from a statistical point of view appear fairly random. The most common method of generating a pseudorandom number is deceptively simple. Given an initial number X_0, we compute all numbers in the sequence using

$$X_i = (bX_{i-1} + c)\%m$$

where b is a large number, c somewhat smaller and m is one greater than the maximal random number. Also, these numbers should be prime numbers or at least *relatively prime*, i.e. not share common divisors. This type of sequence is called a *linear congruential sequence*.

In actual practice, we sometimes do not use a number m. Clearly, if some X_i is greater than the greatest number which the data type can store, it is simply truncated. Unfortunately, this means, in practice, that m is then some power of two, and that tends to make the low order bits periodic. However, it is perhaps slightly more efficient than an explicit modulo operation and it is good practice not to rely too much on the randomness of low order bits in any case.

We often need our random numbers as floating point numbers. The conversion is achieved simply by dividing by $m - 1$ which is the biggest number in the sequence.

[1] Donald E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, volume 2. Addison-Wesley, 1997

4.2 Noise

For a brief moment, one might think that a function which given a random number for a point in space is all it takes, but it is soon obvious that we always need the same random number for the same point in space. Thus, the function needs to be *repeatable*. Thus, a crude noise function that is easy to make simply stores a bunch of random numbers in a 3D (or 2D) table and for a given point it returns the appropriate number.

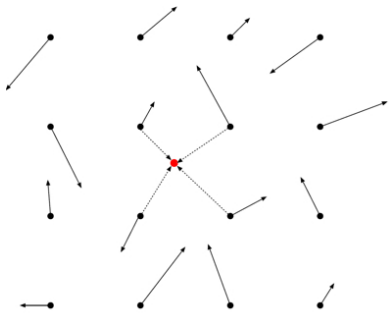


Figure 4.1: A grid of random vectors. The red dot indicates the point $\vec{p}$ where we want to evaluate the noise function.

Some problems with this simple approach immediately pop up. What if the point is outside the domain? Well, in that case we simply repeat the sequence, dividing space into cuboid tiles where each tile has the same pattern of noise. Worse is the fact that we are usually interested in a random number for every single point in $\mathbb{R}^3$ (or our floating point approximation) not just a random number for every integer grid point. This can, of course, be dealt with simply by interpolating numbers on a grid (or lattice – we will use these two words to denote the same) to an arbitrary point in space. This type of noise function is called lattice noise, and it is the simplest type of noise which is probably used in practice. Unfortunately, that is not a very good solution. The reason is that similar random numbers might cluster. We say the noise is not *stationary* if, for instance, the local average is different in some region than in another region.

4.2.1 Perlin's Original Noise

Perlin's original (as well as the improved) noise function handled that by storing not noise itself but random vectors in a 2D or 3D lattice as shown in Figure **??**. To evaluate the noise at a point $\vec{p}$, we find the four lattice points that are at the corners of the grid cell containing $\vec{p}$. We calculate the dot product with each of the four vectors and the corresponding vector from the lattice point to $\vec{p}$. This gives us four numbers which we interpolate smoothly to $\vec{p}$. So, at a grid point the value is ... ? That is right! At any grid the value is zero since at a grid point, we normally interpolate in such a way that only the value at that point counts. At the lattice point, however, the vector to $\vec{p}$ is zero. At first sight that is very strange, but the point soon becomes clear. We want the noise to have the same statistical properties in the entire domain. Forcing it to be zero on grid points is a way of ensuring that it does not have regions where the average is very far from zero (zero is probably in the middle of the range and thereby the average that we should have).

4.2.2 *Frequency Domain Noise*

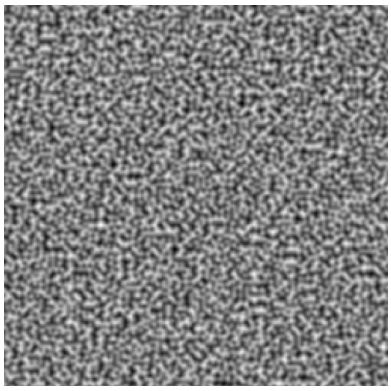


Figure 4.2: An image of noise. This looks quite dull, but we can say a few things: the noise is clearly band-limited, but after close inspection we will have to admit that it is probably not completely isotropic.

Figure **??** shows a block of noise created using the method that was just described. While it works quite well, it is somewhat complicated to describe. Readers familiar with the Fourier transform might have been thinking that what we are trying to achieve is really a band limited noise. Forcing the function through zero at grid points helps avoid very low frequencies in the noise while the lattice limits how high frequencies that we can have, i.e. if we zoom in the interpolation means that the noise starts looking very smooth.

However, thinking about the Fourier transform and the frequency domain quickly leads to a simpler type of noise. Just add some random waves which we can express as follows:

$$w(\vec{p}) = \sin(\vec{f} \cdot \vec{p}) \quad ,$$

where $\vec{f}$ is a random vector. Since we take the dot product, w will produce a wave pattern in the direction of $\vec{v}$ where $\vec{f}$ also gives the frequency. As a small exercise, try to draw $\vec{f}$ and the wave pattern on a piece of paper.

The code for noise is now very simple:

```
float noise(float x, float y) {
  // initialize the first time
        float h=0;
        for(int i=0;i<N;++i)
                h += sin(dot(Vec2f(x,y), f[i]));
        return (1.0/N)*h;
}
```

where f is an array of vectors that has been preinitialized. The way this is done is to generate vectors in $[-1,1] \times [-1,1]$ and throw away vectors that are either longer than unit length or shorter than some other determined length. When N vectors have been reached, we have a set of vectors and a corresponding set of waves that when added produce a noise pattern that does not have very high frequencies (long $\vec{f}$s are thrown away) or low frequencies (we also get rid of short $\vec{f}$). Note that we no longer have special lattice points where the value is zero – except that the value is indeed zero at the origin. That is not an issue, but it could be fixed simply by adding a phase offset to all the sine functions.

In summary, a noise function takes an nD vector as its parameter and returns a number. The defining properties of a noise function are that it is repeatable, i.e. given the same point it returns the same value, and that the values returned seem random (at least when the

vectors given as arguments are not too similar). This definition is not quite sufficient, though. For a noise function to be considered good, we require that it

1. does not exhibit regularity or periodicity.
2. is stationary, i.e. a small patch in one part of the domain is not different from a similar patch in another part.
3. is isotropic, i.e. it does not look different if we rotate it.
4. has known range (e.g. $[-1, 1]$) with the mean in the middle of the range.
5. is band-limited which entails that it is a smooth function. In theory C^∞. In practice, we often settle for C^1.

So far, we did not talk about isotropy. The point here is that the noise function should also be invariant to rotation: the rotated noise should not look different from the original. In practice, that is very hard to ensure when we use a lattice to define the noise, but there should be no anisotropy in the presented, simple noise function that just adds waves.

4.2.3 Turbulence

We have taken great pains to construct noise functions which are band limited. They do not contain very slow or very fast noise components. Now, we want to change that, but it is important that the starting point is a band limited noise. Figure **??** shows three 1D noise images. Observe the image on the left. It does not look noisy. Indeed it is just a smooth bump. Yet, it is the same function that is seen in the middle where we zoom out. The appearance of a noise function depends a lot on scale.

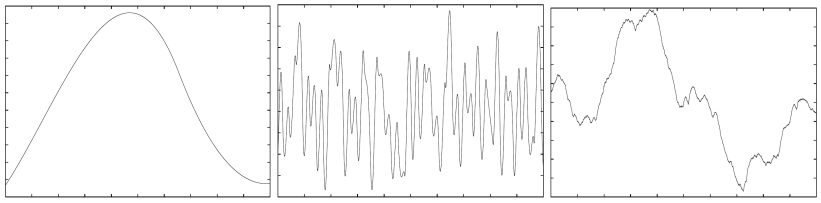


Figure 4.3: From left to right: slow noise, fast noise and turbulence.

In order to create content with our noise functions, we need to achieve something like the image on the right in Figure **??** which looks a bit like a cross section of a very vertical mountain range. That is achieved by adding noise functions at increasing frequency

and decreasing amplitude. Typically, we use the function that Perlin referred to as "turbulence"

$$T(\vec{p}) = \sum_{i=1}^{\infty} \frac{1}{2^i} |N(2^i \vec{p})|$$

where N is our noise function (which could be of any dimension). Notice that with each increment of i, we double the frequency and halve the influence of the noise function. Taking the absolute value means that the gradient field becomes discontinuous. A more complete motivation for the turbulence function was given by Ken Perlin (see the appendix of the cited paper).[2]

The turbulence function gives us something that looks more realistic when creating things like terrain where we have features at many levels of detail but the finer details are much smaller than the coarse ones.

[2] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985

4.3 *Creating Textures and Terrains*

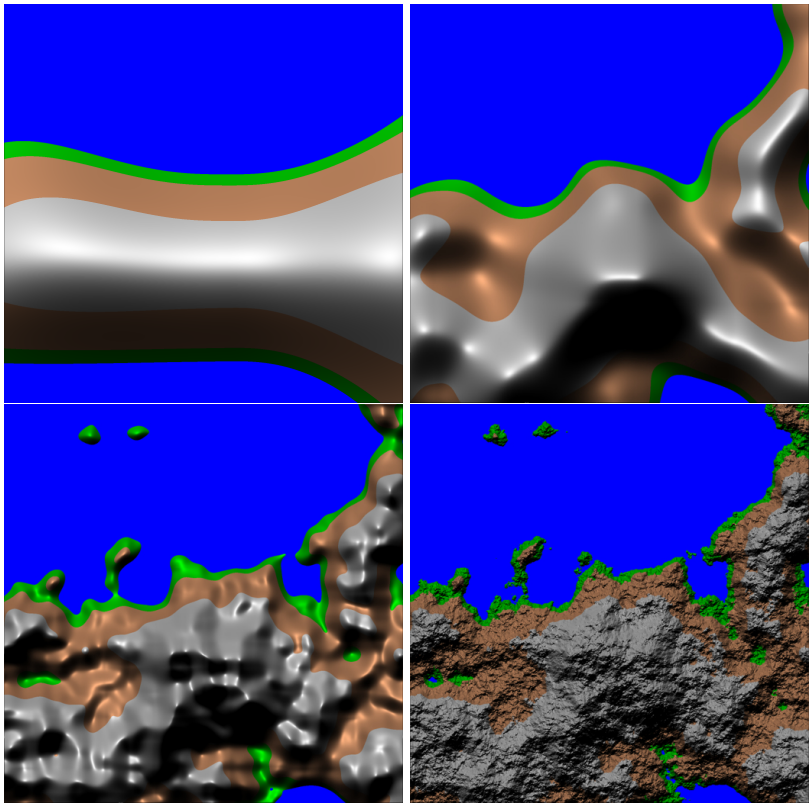


Figure 4.4: A terrain created using a turbulence function with 1, 3, 5, and 15 frequencies of noise.

We can create a procedural terrain in a number of ways, but a simple approach is to sample a turbulence function on a regular grid and directly map the noise value to height. This approach was used for the images in Figure **??**. Of course, we need a little bit more to make the terrain actually look like a terrain. In the example, values less than 0 are mapped to solid blue to indicate water. For height values above zero, the terrain is shaded and the shading is modulated by a green, brown, or grey depending on the height value. Admittedly, the result does not look realistic. Noise could have been used in several more ways to improve things. For instance, we could have added a bit of noise to the height value before doing color lookup. We could also have added noise to the result of the color lookup. Both of these things had added a bit of natural variation which would have made the terrain look more plausible.

Figure **??** shows terrain in a real-time application where we have added trees and a grass texture as well as a gradient sky and reflecting water. The point, however, is that in all three images, we use the same noise function, but only in the middle image is the noise mapped directly to height. In the top image, we map using the square root, and in the bottom we take the square. Thus, we can shape the noise function easily by using other functions to map it.

Ken Perlin introduced a general scheme for mapping noise or turbulence, namely the so called bias

$$\text{bias}_\alpha(x) = x^{-\frac{\ln \alpha}{\ln 2}},$$

and gain

$$\text{gain}_\alpha(x) = \begin{cases} \frac{1}{2}\text{bias}_\alpha(2x) & x < 0.5 \\ 1 - \frac{1}{2}\text{bias}_\alpha(2 - 2x) & x \geq 0.5 \end{cases}$$

These functions have complex expressions, but their purpose is simply to either bias the function towards high or low values or, in the case of gain, to polarize the function so that values move towards the extremes or the middle. See Figure **??**.

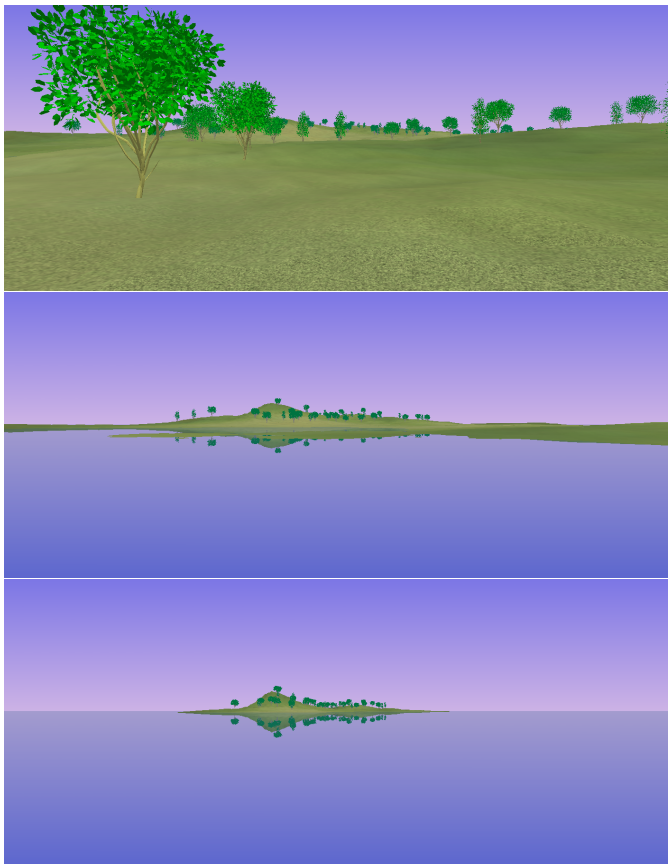


Figure 4.5: Three different terrains created from the same height function. Top: the sqrt of the height function was used to generate the terrain. Middle: the height was used unmodified. Bottom: the square of the height function was used.

The final tools needed for effectively creating textures are a base function and color mapping. See Figure **??**. On the left a simple 2D saw tooth pattern is shown. On the right it has been transformed into a 2D wood texture. Initially, the saw tooth was deformed by adding noise to the lookup point, turbulence was added to the value, and the whole thing was mapped to brownish shades. The result is not a convincing wood texture, but maybe a passable one for simple purposes.

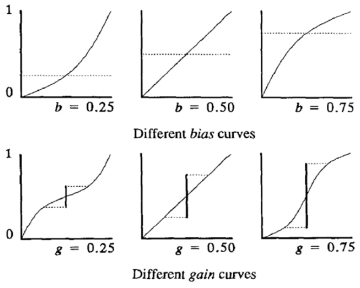


Figure 4.6: The bias and gain curves

4.4 *Hypertexture: Creating Fur and Stuff*

Creating 2D textures is not the only possibility. Another one of the original ideas Perlin had was to create *hypertexture*. Essentially, a hyper texture is a 3D texture created using much the same tools as for 3D texture generation. In figure **??**, a 3D cloud is shown. It was created by initially generating a function that is 1 inside an ellipsoid and 0 outside. In a narrow region around the surface of the ellipsoid

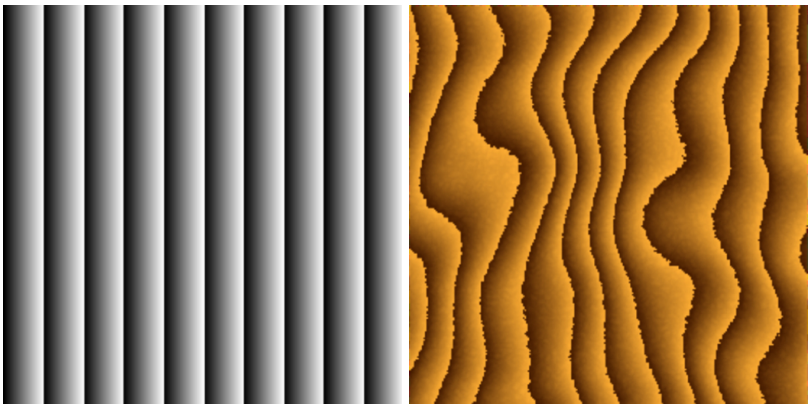


Figure 4.7: Left: a 2D saw tooth pattern. Right: a wood texture created using noise, turbulence, and color mapping.

it goes from 0 to 1. In the leftmost image we just a volume rendering of the cloud. In the middle image, noise and turbulence have been added to the lookup position and, in the right image, the intensity is mapped to color and alpha. Even full intensity is mapped to a very low alpha value to give the cloud a translucent appearance. The

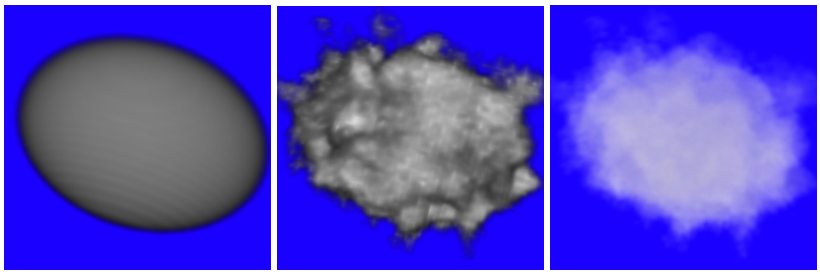


Figure 4.8: A cloud is created from a distance field of an ellipsoid. Left: the unmodified distance field is volume rendered. Middle: Noise and turbulence are used to warp the distance field. Right: mapping the distance field to color and (very low) alpha value results in a somewhat cloudy appearance.

idea of having a 3D texture represent an object is very general. Usually we think of the intensity as a proxy for material density, and instead of mapping the intensity to alpha we can also extract a surface. Typically we choose some threshold level, say 0.5, and the isosurface or level set where the volume has intensity exactly 0.5 becomes the extracted surface. This *volumetric* object representation also lends itself well to very simple tools for editing. The volume representation and interactive editing is something that we will return to in the next chapter.

Hypertexture could also be used the create things like fur. However, it is difficult to use hypertexture in the original formulation in real-time graphics. Instead, a common method for fur visualization is to render a stack of slices for each furry polygon where each slice represents a cross section of the fur tufts. Thus, we use a simple texture which has high transparency where there is air and low trans-

parency where we need fur. When stacked we get the appearance of tufts of fur sticking out of the model as shown in Figure **??** (left). On the right in the same figure, we see a fur covered arch. The method is useful for fuzzy things like a teddy bear, where the fur is supposed to be a bit like on a brush. The method cannot be used for long or curly hair.

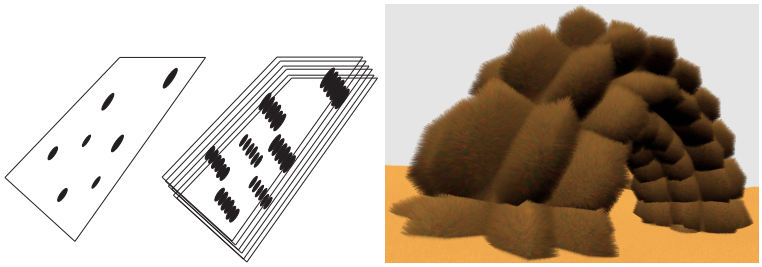


Figure 4.9: Left: Fur rendering principle. The fur is rendered by having multiple fur slices rendered on top of each other. On the far left, we see a single slice and then a stack of slices rendered on top of each other. Right: A fur covered arch.

Unfortunately, we cannot use a geometry shader to generate all the fur slices in one go. That would be very convenient, but to get the compositing right, we need to render the slices closest to the object first, and then progress outwards.

4.5 *Creating Trees and Plants*

The things we can create almost exclusively by mapping a turbulence function are mostly restricted to terrain, texture, and amorphous things like clouds. For objects with a hard and well defined surface, we generally use grammar based methods. For botanical objects, the usual method is *L-systems*. The following discussion is based on a description by Prusinkiewicz et al.[3] and follows that paper quite closely, although the paper also covers many advanced methods that we do not go into here.

[3] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomir Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer, 1996

An L-system is a simple set of grammatical rules, known as *production rules*, which allows us to rewrite a text string. A simple example of a production rule is

$$A \rightarrow AB$$

where the symbol to the left is rewritten as shown on the right. Given an initial string A we obtain

$$A \rightarrow AB \rightarrow ABB \rightarrow ABBB \rightarrow \quad ABBBB$$

after 1, 2, 3, and 4 rewrites using the production rule.

Turtle graphics. If we associate some geometric meaning with each symbol, we start being able to use this for computer graphics. However, we also need to be able to associate parameters with each symbol. Suppose that we have the following string

```
F(1) +(-120) F(1) +(-120) F(1)
```

where *F* is move forward (while drawing), + means turn counterclockwise, and the numbers in parentheses are parameters. Thus, the above string means "move forward 1, turn 120 degrees clockwise, move forward 1, turn 120, move forward". Observe that the result is a triangle if we interpret the string as drawing instructions.

Now, we can make a production rule that replaces each (draw) forward command with something more:

```
F(s) → F(s/3) +(60) F(s/3) +(-120) F(s/3) +(60) F(s/3)
```

Running a single rewrite of our triangle producing string gives this

```
F(1/3) +(60) F(1/3) +(-120) F(1/3) +(60) F(1/3)
+(-120)F(1/3) +(60) F(1/3) +(-120) F(1/3) +(60)
F(1/3) +(-120) F(1/3) +(60) F(1/3) +(-120) F(1/3) →
+(60) F(1/3)
```

It may not be immediately obvious, but this is the first iteration of an algorithm that produces the Koch's snowflake. A few more iterations are shown on the right in Figure **??**.

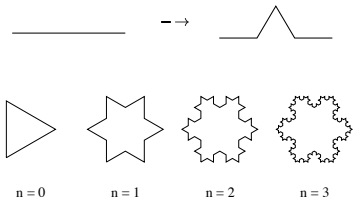


Figure 4.10: Three iterations of a production that results in the Koch snowflake fractal

We can go on to draw a 3D object by extending the vocabulary a bit. First, we need to take a closer look at how we convert the string to a 3D model or a drawing. In 2D, we can use the notion of turtle graphics: above we have tacitly assumed that the turtle has a position and is pointing in *some* direction. Let us call that direction Y. F(x) simply means that the turtle should draw a line segment of length x from its current position in the direction of Y.

If we think of Y as the Y axis and assume that Z points up and X to the right, the symbol + corresponds to rotation around the Z axis (pointing out of the screen). If we want to draw in 3D, it becomes convenient to be able to rotate around the other axes too. In the following, we only need rolling, i.e. rotation around the Y axis (the direction in which we head). Specifically, we will use the symbol /(a) for a rotation of a degrees around the Y axis.

In summary, our turtle carries a frame of reference. It always moves in the Y direction while drawing, and it can rotate around the Z axis as well as the Y axis. So far, we can only go forward and rotate. It seems that all the turtle can draw is a sequence of line segments. Two commands vastly increase our vocabulary. Push which

has the symbol '[' causes the turtle to push its current state onto a stack. The state is primarily the frame of reference. To record that, we push the matrix used to transform the frame of reference onto the stack. The other command is Pop which has the symbol ']'. Popping simply copies the information at the top of our stack into the current state. What these two commands provide is the ability to create branching structures. If we are drawing the trunk of a tree and push, then draw a branch, we can now pop to get back to the branch point and continue along the trunk or draw another branch.

We are now ready to do a tree. Prusinkiewicz et al. proposed an L-System for tree generation which is surprisingly powerful and versatile and contains just a single production rule

$$
\begin{aligned}
A(s,w) : s \geq s_{\min} \rightarrow\ & !(w)\ F(s) \\
& [+(\alpha_1)/(\phi_1)A(s\, r_1\ ,\ wq^e)\ ] \\
& [+(\alpha_2)/(\phi_2)A(s\, r_2\ ,\ w(1-q)^e)\ ]
\end{aligned}
$$

where α_1, α_2, ϕ_1, and ϕ_2 are angles used for turning and rolling the turtle, respectively. s and w represent the length and width of (a branch of) the tree, and r_1, r_2, q, and e are constants used to scale s and w. $s \geq s_{\min}$ is the *condition* for applying the rule. Thus, we stop growing when the length of new branches falls below a certain threshold. The A symbol is not visualized but encapsulates the s and w parameters which are thus stored in the L-System string. The '!' symbol sets the pen width when the turtle draws.

What the rule does is to repeatedly replace the A symbol with a line segment and then create a bifurcation with two new branches also signaled by A symbols. The process mimics growth. The A symbols represent growth zones and when they are removed, they leave behind line segments. If we start from A(100,20) and run the production rule to level 5 with the parameters corresponding to Figure 8g in Prusinkiewicz et al.[4] the results are as shown below:

```
!(20)F(100)[+(30)/(137)!(14.1421)F(80)[+(30)/(137)!(10)
F(64)[+(30)/(137)!(7.07107)F(51.2)[+(30)/(137)!(5)
F(40.96)[+(30)/(137)A(32.768,3.53553)][+(-30)/(137)
A(32.768,3.53553)]][+(-30)/(137)!(5)F(40.96)[+(30)
/(137)A(32.768,3.53553)][+(-30)/(137)A(32.768,3.53553)]]]
[+(-30)/(137)!(7.07107)F(51.2)[+(30)/(137)!(5)F(40.96)
[+(30)/(137)A(32.768,3.53553)][+(-30)/(137)
A(32.768,3.53553)]][+(-30)/(137)!(5)F(40.96)[+(30)
/(137)A(32.768,3.53553)][+(-30)/(137)A(32.768,3.53553)]]]]
[+(-30)/(137)!(10)F(64)[+(30)/(137)!(7.07107)F(51.2)
```

[4] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomir Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer, 1996

Figure 4.11: The same tree (identical L-System strings) but converted from L-Systems to polygons in two different ways. On the right, the geometry has been created simply using cylinders. On the left, the SQM method was used to convert the tree into a coherent mesh.

[+(30)/(137)!(5)F(40.96)[+(30)/(137)A(32.768,3.53553)]
[+(−30)/(137)A(32.768,3.53553)]][+(−30)/(137)!(5)F(40.96)
[+(30)/(137)A(32.768,3.53553)][+(−30)/(137)
A(32.768,3.53553)]]][+(−30)/(137)!(7.07107)F(51.2)[+(30)/(137)
!(5)F(40.96)[+(30)/(137)A(32.768,3.53553)][+(−30)
/(137)A(32.768,3.53553)]][+(−30)/(137)!(5)F(40.96)[+(30)
/(137)A(32.768,3.53553)][+(−30)/(137)A(32.768,3.53553)]]]]]
[+(−30)/(137)!(14.1421)F(80)[+(30)/(137)!(10)F(64)[+(30)/(137)
!(7.07107)F(51.2)[+(30)/(137)!(5)F(40.96)[+(30)/(137)
A(32.768,3.53553)][+(−30)/(137)A(32.768,3.53553)]]
[+(−30)/(137)!(5)F(40.96)[+(30)/(137)A(32.768,3.53553)]
[+(−30)/(137)A(32.768,3.53553)]]][+(−30)/(137)!(7.07107)
F(51.2)[+(30)/(137)!(5)F(40.96)[+(30)/(137)A(32.768,3.53553)]
[+(−30)/(137)A(32.768,3.53553)]][+(−30)/(137)!(5)F(40.96)
[+(30)/(137)A(32.768,3.53553)][+(−30)/(137)A(32.768,3.53553)]]]]
[+(−30)/(137)!(10)F(64)[+(30)/(137)!(7.07107)F(51.2)[+(30)/(137)
!(5)F(40.96)[+(30)/(137)A(32.768,3.53553)][+(−30)/(137)
A(32.768,3.53553)]][+(−30)/(137)!(5)F(40.96)[+(30)/(137)
A(32.768,3.53553)][+(−30)/(137)A(32.768,3.53553)]]]
[+(−30)/(137)!(7.07107)F(51.2)[+(30)/(137)!(5)F(40.96)
[+(30)/(137)A(32.768,3.53553)][+(−30)/(137)

A(32.768,3.53553)]][+(−30)/(137)!(5)F(40.96)[+(30)/(137)
A(32.768,3.53553)][+(−30)/(137)A(32.768,3.53553)]]]]]

while the above looks like complete gibberish to a human being, it is very easy for a computer to interpret. However, we would like not to simply draw a black line in space but to produce a triangle mesh that represents a tree. In the source code that accompanies the exercises, we have created an explicit turtle class. All the L-system commands that correspond to transformations ('+', '/', '!', '[', and ']') are used to transform the turtle. When a drawing command F(s) is encountered, we draw a truncated cone of length s and simultaneously move the turtle forward s units.

With this in place, our program can spit out the tree in Figure **??** (right). One immediately obvious shortcoming is that the branches do not blend nicely but join at sharp angles - unlike the tree on the left. The tree on the left was created by converting the string representation to a graph and then using the SQM (Skeleton to Quad-dominant Mesh) graph to mesh conversion method by Bærentzen et al.[5] to generate a water tight polygonal mesh from the graph representation.

Figure **??** shows two trees that are both created using the very same L-system rule - just with different parameters.

[5] Jakob Andreas Bærentzen, Marek Krzysztof Misztal, and K Wełnicka. Converting skeletal structures to quad dominant meshes. *Computers & Graphics*, 36(5):555–561, 2012

Figure 4.12: These two trees are created using L-Systems. In fact, the same rule is used with different parameters.

4.6 Learning More

Pseudo-random numbers is an important aspect of procedural synthesis. Donald Knuths second volume is an authoritative and also entertaining place to read more about linear congruential sequences which is what we generally use. Using noise for creating 3D models and textures is covered broadly by Ebert,[6] but we also recommend Perlin's seminal paper.[7] While the paper which we relied on for the L-system section is a good starting point,[8] the Algorithmic Beauty of Plants by Prusinkiewicz and Lindemayer is the authoritative reference,[9] containing many details about a variety of aspects related to plant growth.

[6] David S Ebert, F Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 3rd edition, 2002

[7] Ken Perlin and Eric M Hoffert. Hypertexture. *Computer Graphics*, 23(3):253–262, 1989

[8] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomir Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer, 1996

4.7 Project: Procedural Texture and Trees

Objectives

Your first goal is to add a procedural texture to the terrain. Your second task is to implement an L-system generator (or rather fix an existing implementation) in order to add a tree to the scene.

[9] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S Hanan, F David Fracchia, Deborah R Fowler, Martin JM de Boer, and Lynn Mercer. *The Algorithmic Beauty of Plants*. Number 6. Springer-Verlag, 1990

Part 1

In the `terrain.frag` shader, there is a sampler called `noise_tex`. In the terrain rendering function `Terrain::draw` in `Terrain.cpp`, you should create a noise texture and bind that texture to the sampler. The noise texture may be simply a texture of random values. In the terrain.frag shader, combine the height value and the noise value in order to create a pleasant procedural texture for the terrain. Try adding noise at several scales.

Part 2

In `lsystem.cpp` there is a function `Rule::apply` for a DoL system, but the production rule is trivial. Implement the production rule used to produce the trees in Figure 8 of Prusinkiewicz et al.[10] (Note that angles are given in degrees in that reference). Explain all of the parameters which are used in that production rule. In the Turtle

[10] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomir Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer, 1996

class, most of the functions are not implemented. Implement the turtle functions `turn`, `roll`, `move`, `push`, `pop`, `set_width`.

Part 3

The interpret function contains a switch statement used for interpreting L-System elements as drawing commands for the turtle. Only `LS_DRAW` is actually implemented. Implement cases for the other L-System symbols. When these three changes have been made, you should be able to see a tree like that shown in Figure **??**, but the example, `ex`, does not produce a pleasing tree. Use the parameters for the tree in Fig. 8g from Prusinkiewicz et al. and parameters for two other trees from the same figure. These parameters are in Table 1 of Prusinkiewicz et al. Finally, tweak both parameters and the actual rule to make your own addition to the world of botany.

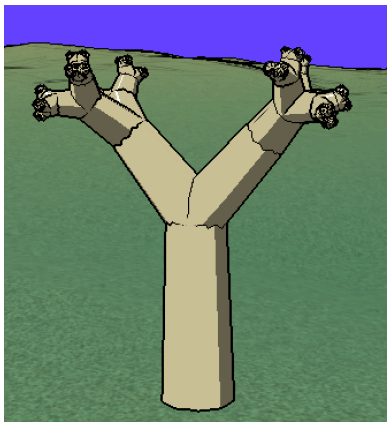


Figure 4.13: An L-System Generated Tree where the rule leaves a lot to be desired.

Deliverables

In part 1, you need to explain what you did, answer questions, and submit images of before and after the changes. We also need the shader code. In part 2, you need to submit the source code for changed functions as well as your own L-system rule and screen shots of the tree generated with your own rule as well as those which reproduce Figure 8 in Prusinkiewicz et al

5
Volumetric Geometry

Sometimes a representation or a method emerges that is so generic it seems poised to become the grand unifying technique for something or other. After a few years of intense hype, its flaws and shortcomings become apparent, however, and people turn away from the supposed panacea only – after more time passes – to accept it as one useful tool among many. I think *voxels* and *volume graphics* fit the bill pretty well.

The notion of a volume is a straight forward generalization of an image. Whereas an image is composed of 2D pixels, a volume is composed of 3D voxels, and there is an equally profound confusion regarding what these two notions really are. After all, is a pixel a point sample in an image or a small square on the monitor? Probably both. The same is true of a voxel. It is usually safest to think of a voxel as a point in a regular 3D grid, but often the same term is used for a cuboid or rectangular cell in a regular grid. Hedging our bets, let us think of a voxel as a point with an associated attribute, say color or intensity, and an associated cell. The voxel notion is illustrated in Figure 5.1.

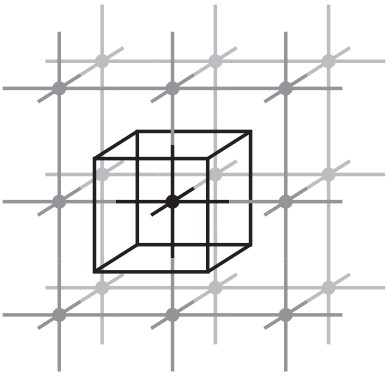


Figure 5.1: A part of a volume (or voxel grid) where the voxels are the grid points (shown as filled circles). Sometimes, we also think of the box enclosing the part of space closest to the grid points as "the voxel". Voxels typically also have attributes such as intensity or color.

This sort of discretization of space is just as powerful as an image, and it allows us to model 3D scenes by the 3D analogue of painting. If we see empty space as white voxels and object as black voxels, the idea should be pretty clear. Just as in the case of an image, however, 3D volumes are subject to *aliasing* – the pesky phenomenon from signal analysis which causes Moire patterns and jaggies. Just as in the image case, we need the signal to be smooth: in other words, the voxel values should be grey level and not binary. Unless, of course we are going for the Minecraft look which, at the time of writing, is quite popular thanks to the eponymous computer game.

While Minecraft made voxels cool, volume data is by no means used only for representation of objects. A volume can also be used to store illumination, and several succesful real–time global illumination

methods use voxels to store illumination information. However, in this chapter, we focus on volumes as a means for storing scene geometry. In a later chapter, we shall return to volumes in the context of illumination.

Initially, in the following section, we discuss how to synthesize a volumetric 3D scene representation. This is followed by a chapter on interactive editing. Interactive editing is quite straight forward in the case of volume data, and that is one of the merits of the representation. It is also very easy to create holes in objects, so voxels lend themselves to creating caves in 3D scenes. The last half of the chapter is concerned with rendering of volumetric scenes. Specifically, we discuss how to convert volume data intro triangle meshes.

5.1 *Introduction to Volume Data and Distance Fields*

Perhaps the simplest way to see a voxel is as a small brick, but if we construe a voxel as a sample of a smooth function, we can avoid the aliasing issues (the brickiness) and use voxels to model smooth surfaces. In this case, we often talk about volumes as *distance fields*. According to this notion, the value of a voxel is the shortest distance to the closest surface, positive if outside and negative inside. This is shown in Figure 5.2. For a distance field, it is pretty clear that the surface is the set of points where the distance to the surface is zero. Unfortunately, since we only know the values of the distance field at the voxels (i.e. the grid points), it is unclear how we should get back the smooth surface. This is where *interpolation* comes to the rescue. Given a function known at a set of grid points, interpolation is the process of reconstructing the value at arbitrary points in space. Interpolation allows us to get back (an approximate) value of the distance field for arbitrary points in space and with this we can reconstruct the original surface as the *level zero isosurface* (aka level set), i.e. the set of points where the function is zero: $S = \{\vec{x} | f(\vec{x}) = 0\}$. Below we discuss trilinear interpolation in more detail.

In many cases, it is a bit constraining to enforce that the voxel value is the actual distance as long as the value changes smoothly close to the surface. If the value is not exactly a distance, we sometimes call it the *algebraic distance* or simply talk about a *scalar field* instead of a distance field.

Distance fields are often generated by sampling functions. For instance, a sphere can be represented implicitly by the equation $S = \{\vec{x} | f(\vec{x}) \leq 0\}$ where $f(\vec{x}) = \|\vec{x} - \vec{x}_0\| - r$ and where $\vec{x}_0$ is the center of the sphere, and r is its radius. Thus, for any point where f is less than zero we are inside the sphere, and the sphere itself is

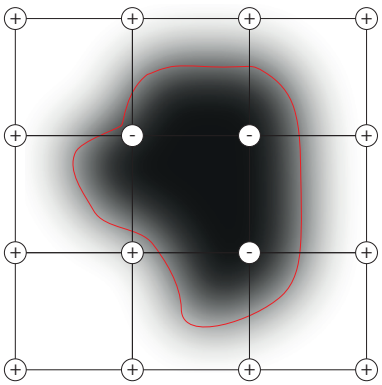


Figure 5.2: A simple object (red) with its distance field shown going from black (negative) to white (positive). Note that the value is not necessarily literally the distance and that we also clamp the distance to a minimum and a maximum value.

the set of points where $f = 0$. Clearly, this simple concept holds for other objects than just spheres although it is not always easy or even possible to find a closed form function f that represents a given object (let alone as an actual distance field). However, this is precisely why voxels are so useful. Given a base object (such as the sphere) we can sample the corresponding function f on a voxel grid and then apply any modifications to the voxel representation – for instance by adding many functions together simply by adding their values at the grid points.

Of course, one might reasonably ask why we do not simply store the functions and add them together for each point in space. The answer, of course, is that sometimes the function evaluations are simply too costly. Another objection is that whenever we use a voxel grid, we really only know the function at a grid of points in space.

5.1.1 *Trilinear Interpolation*

Sometimes, we deal with voxel data on the CPU and need to implement linear interpolation ourselves. The formula for linear interpolation was introduced in the first chapter of this book and illustrated in Figure 2.9 for the simplest case where we just interpolate between two points in 1D. Let us first consider how to extend this to bilinear interpolation in 2D where we need to interpolate between the four corners of a square as shown in Figure 5.3. As the figure indicates, bilinear interpolation can be expressed as three linear interpolations: we first interpolate twice to get the values at the green dots and then one more time to get the desired value at the blue dot.

$$\begin{aligned} v^{\text{blue}} &= (1-\beta)v_0^{\text{green}} + \beta v_1^{\text{green}} \\ &= (1-\beta)((1-\alpha)v_{00} + \alpha v_{10}) + (\beta)((1-\alpha)v_{01} + \alpha v_{11}) \end{aligned} \tag{5.1}$$

where the values α and β range from 0 to 1. Thus, to get the interpolation weights (α, β) for a point $\vec{p}$, we subtract the position of v_{00} from $\vec{p}$ and divide by the distance between adjacent grid points.

In 3D, we have three weights α, β, and γ, and we need to interpolate between eight voxels instead of four pixels, but the principle is the same. Now, we just need a total of seven linear interpolations: four along the first axis, two along the next, and one along the final axis.

To improve efficiency, we can rewrite the interpolation as a weighted sum of the eight voxels where each weight is a product of α or $1-\alpha$, β or $1-\beta$, and γ or $1-\gamma$.

We also need to take into account that the position given for in-

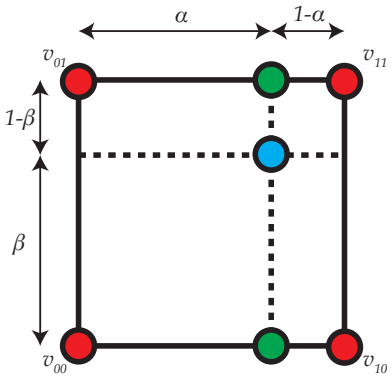


Figure 5.3: This figure shows bilinear interpolation. The α weight is used to interpolate horizontally between v_{00} and v_{10} and then v_{01} and v_{11} producing values at the two green points. β is then used to interpolate vertically between the green points, producing the value at the blue point.

terpolation might be outside the domain. In many cases, the best strategy for dealing with this issue is to project the position back inside the domain. Inside rather than just onto the boundary of the domain since we must ensure that our code does not try to access voxels outside of the grid.

Putting these things together, we obtain the C++ code snippet below. Note that the code snippet assumes that no scaling is required.

```
// Code for trilinear interpolation
float interpolate(const RGrid<float>& grid, const Vec3f& _p)
{
    Vec3f p = v_min(Vec3f(grid.get_dims()) - Vec3f(1+1e-6),
                    v_max(_p, Vec3f(0,0,0)));
    Vec3i po(p);
    Vec3f a[2];
    a[1] = p-Vec3f(po);
    a[0] = Vec3f(1.0)-a[1];
    float f = 0;
    for(int i=0;i<8;++i) {
            int xbit = i&1;
            int ybit = (i&2)>>1;
            int zbit = (i&4)>>2;
            const Vec3i corner = po+Vec3i(xbit,ybit,zbit);
            f += a[xbit][0]*a[ybit][1]*a[zbit][2]*grid[corner];
    }
    return f;
}
```

5.2 *A World of Voxels*

The volume representation can be used for many purposes, and the topic of voxelization, turning 3D geometry from, say, a triangle mesh into a voxel representation, is a fairly big topic. We have already discussed the fact that given an implicit representation, we can simply sample the function, f, at grid points. Given a triangle mesh, we tend to compute the signed distance from each voxel to the closest point on the mesh, producing a signed distance field. However, these topics are a bit beyond the scope of the current chapter. Instead, we will focus on creating a voxel based 3D landscape. In fact, we restrict the scope even further and consider only how we can represent the landscape geometrically. It is quite possible to go further by storing a material attribute per voxel, thereby providing information about the materials inside the volume which could potentially allow us to create a geological model for the world.

Now, if we just want a flexible terrain model, we might as well start with a height field. In Chapter **??**, we discussed the use of noise and turbulence functions for precisely the purpose of generating height fields. A single ingredient that we might want to add is the use of a base height function. If we want to create a terrain with a particular overall shape, the base height function can be used to define this shape and turbulence can be used to add detail. Thus, our procedural terrain height field h might be defined thusly:

$$\begin{aligned} h(x,y) &= k_0 \exp(-(x^2+y^2)/\sigma_0^2) \\ &+ k_1 \exp(-(x^2+y^2)/\sigma_1^2) T(sx, sy). \end{aligned} \tag{5.2}$$

This is clearly a Gaussian plus a turbulence function T scaled by another Gaussian, where k_i are constants that control the relative amplitude of the two terms, and the σ_i are constants that control the shape of the Gaussians. Finally, s controls the size of the turbulence.

Unfortunately, this is not a volumetric terrain, but it is very easy to make it so: we simply subtract the height value from z:

$$f(x,y,z) = z - h(x,y). \tag{5.3}$$

This approach works well, but we might have to spend some time tweaking the several parameters of the above terrain model. Clearly, f is not the distance to the closest point on the surface but the perpendicular distance to the surface, but, in practice, this does not make any difference. The zero level set is still the initial height field. We might want to clamp the function, though. Whenever we are far from the zero level, the actual voxel value is not of interest:

$$f_{\text{clamped}}(x,y,z) = \min(c, \max(-c, f(x,y,z))), \tag{5.4}$$

where c is the value to which we clamp. Often a small integral number such as 2 is a good idea assuming we have a voxel grid spacing of 1. This ensures that the value is not clamped in the immediate vicinity of the surface.

Unfortunately, the terrain does not exploit the fact that we are using voxels, but we are now ready to take the next step. We can create a porours, cave filled terrain, simply by subtracting noise:

$$f_{\text{porous}}(x,y,z) = f_{\text{clamped}} + k_2 |N(s_1 x, s_1 y, s_1 z)|. \tag{5.5}$$

where k_2 and s_1 scale the size and the amplitude of the noise function N, respectively. The reason for adding the absolute value of the noise is that we do not want to create spurious structures outside the terrain: we only want to create holes inside the terrain. Adding the absolute value of the noise means that positive regions can only become more positive but negative regions can change sign, thereby

going from being interior to being exterior. Note that if the original terrain function had not been clamped, the noise amplitude needed to make caves would have been different depending on how far inside the terrain we are.

To produce the final terrain, all we need to do is to sample (5.5) at all voxels in our volume.

$$\text{Foreach } i,j,k \in G : G[i,j,k] \leftarrow f_{\text{porous}}(i,j,k) \tag{5.6}$$

where G is the voxel grid and i, j, k are indices of actual voxels and thus $[i,j,k]$ is also the integer position of the voxel in the grid.

In many cases, we want a grid that is of greater dimension in X and Y than Z. The example shown in Figure 5.4 utilizes a grid size of $512 \times 512 \times 64$. It is clearly a very porous terrain with numerous cave mouths, precisely as intended.

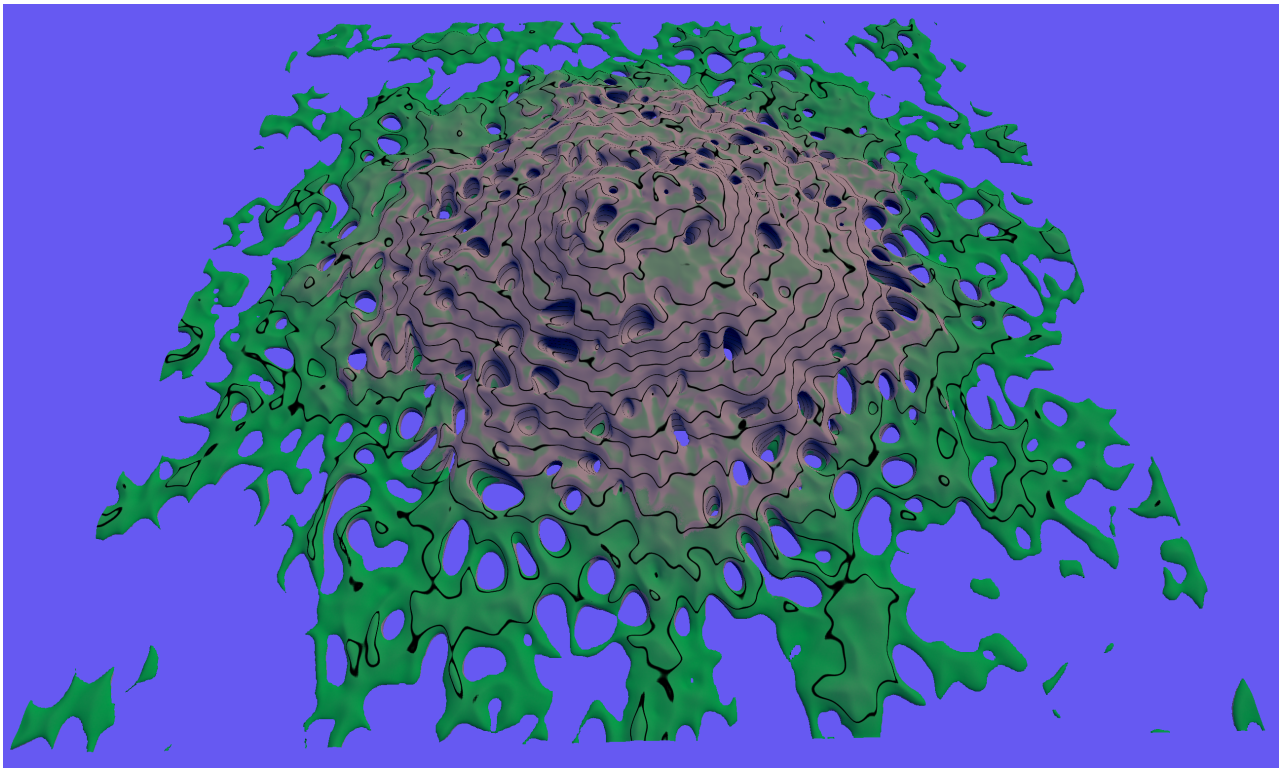


Figure 5.4: This figure shows a synthesized terrain. Several elements have been combined here. A height map was initially combined with turbulence and turned into a volumetric terrain. Then caves were added with a 3D noise. A procedural terrain color synthesizer was combined with rendering of height curves.

5.2.1 *Using all your Cores*

We often go to great lengths to utilize our GPUs. There is nothing wrong with that, but we should also remember that (as of writing)

any computer (and many smartphones) sold will have at least two cores and probably many more, especially if we count virtual cores due to hyper threading. The upshot is that it is often much more efficient to run code in, say, eight parallel threads or more than it is to run single threaded.

Terrain generation requires many operations that are carried out independently for each voxel. Thus, voxelization is often embarrasingly parallel: it is possible to compute the value of each voxel independently from that of any other voxels. Such problems are very easy to parallelize because no synchronization is needed.

Unfortunately, if we need the terrain on the CPU it will incur substantial overheads in code complexity and data transfer to perform the generation using the GPU. On the other hand, multithreaded code that does not require locks or synchronization is increasingly simple to write. For instance, the C++11 standard introduced threading primitives as a part of the programming language, and it often takes only a few extra lines of code to vastly speed up computations using these threading primitives. To be more specific, when running the terrain generation for the example in Figure 5.4 the program utilizes over 750% of a single core. Thus, we obtain a speed up that is quite dramatic compared to a single threaded program that could not run at more than 100% utilization.

5.3 *Interactive Editing*

Interactive editing is the whole motivation for using voxels. With the volume representation, we can easily add and remove structures from the volume in a local fashion. Thus, the editing operations almost always proceed by first identifying the region of the volume that needs to be edited and then performing the edit operation itself.

Identifying the region is not completely trivial. In a game-like scenario, the position of our avatar (user, player whatever) could be a first approximation of where we want to make the edit, but often we prefer to use a point that is close but in front of the character. Finding a precise point is easily done by tracing a ray through the volume. The code snippet below illustrates a simple way of tracing rays.

```
Vec3f ray_trace(const RGrid<float>& grid, const Vec3f& po, const Vec3f& dir) const {
    Vec3f p = po;
    const float step = 0.1;
    while(interpolate(grid,p)>0 && grid.in_domain(Vec3i(p)))
        p += step*dir;
    return p;
}
```

It can be done more efficiently in several ways: we can enumerate the grid cells (i.e. the cells whose corners are voxels) which are pierced by the ray and only check a cell for intersection once. We can also use a bigger time step and then use bisection to find a very precise intersection point. However, these optimizations add complexity, and if we only trace very few rays, the run time of the ray tracing procedure might be completely irrelevant to the overall performance.

Having found a point $\vec{p}$ at which we wish to center the local edit operation, we need to decide the size of the region that is affected and also to ensure that there is a smooth transition between the edited region and the rest of the volume. This requires that we have a *transition region*: a layer of voxels inside the edited region. Going across the transition region from the edited region to the outside, we need the effect of the tool to fall off smoothly from full effect to no effect.

Let us denote the operation that updates the volume g. We can use several types of *falloff*, but it is simple to employ the smooth step function

$$\begin{aligned} \mathrm{ss}(a,b,x) &= \left(\tfrac{x-a}{b-a}\right)^2 \left(3 - 2\left(\tfrac{x-a}{b-a}\right)\right) \\ &= 3t^2 - 2t^3, \end{aligned} \tag{5.7}$$

where $t = \frac{x-a}{b-a}$. This function, which is a standard tool in GLSL, has the property that its derivative is 0 at both $t = 0$ and $t = 1$ and it goes smoothly from 0 to 1. Clearly, we need to test whether t is in the range, and this is easier to show in pseudocode:

```
float smooth_step(float a, float b, float x)
{
    float d = b-a;
    float t = (x-a)/d;
    if(t<0.0) return 0;
    else if(t>1.0) return 1;
    else return t*t*(3 - 2*t);
}
```

Given a region of interest centered at $\vec{p}_0$ with radius R, we can now express the update operation as follows:

$$\begin{aligned} \text{Foreach} \quad i,j,k \quad &\in \text{ROI}: \\ \alpha \quad &\leftarrow \quad \mathrm{ss}(R, R-T, \|\vec{p} - \vec{p}_0\|) \\ G[i,j,k] \quad &\leftarrow \quad \alpha g(i,j,k) + (1-\alpha) G(i,j,k) \end{aligned} \tag{5.8}$$

where $\vec{p} = [i,j,k]$ and $\vec{p}_0 = [i_0, j_0, k_0]$ and T is the width of the so called transition region. In other words, we perform a weighted average for each voxel of the old value (already in G) and the new value (provided by g) and the weights sum to 1. Note that the tool

affects a spherical region since we plug the distance to the center of the tool into the smooth step function. Typically the ROI is a cuboid region since it is only easy to loop over such a region and the smooth step function will be zero outside of the circumscribed spherical region anyway.

5.3.1 *Adding and Subtracting*

We can add and subtract material quite easily by using a constant function

$$g(i, j, k) = m$$

where m is the material constant. Clearly, if we assume that the volume is negative inside and positive outside, then using $m > 0$ we will create a small hole by applying (5.8) and if $m < 0$ we add material. The result of adding for various sizes of transition regions is shown in Figure 5.5.

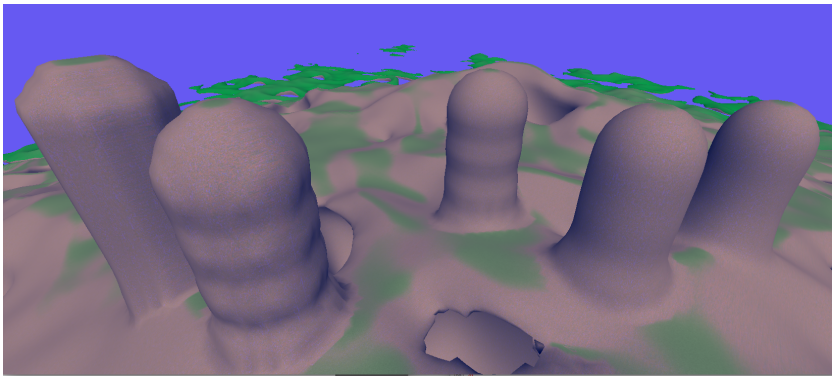


Figure 5.5: Pillars created by adding spherical elements on top of each other. From left to right, the width of the transition region is 0, 1, 2, 3, and 4. Notice how the grid structure becomes apparent when we use no transition region and how the spheres blend much more smoothly for very thick transition regions.

There is a lot of potential in this simple operation. Using tools of different shapes we can perform volumetric boolean operations. It is also possible to select a part of the volume, copy it and paste it elsewhere.

5.3.2 *Smoothing*

Often we want to smooth a surface. This is quite straightforward in the case of volume data. We simply smooth (or blur) the voxels, and the resulting isosurface will become smoother. Again, we usually want a gradual transition between the smoothed region and the rest of the volume. To attain the desired effect, we only have to replace

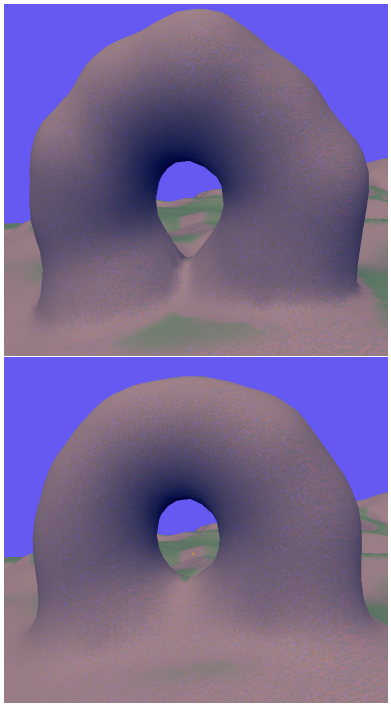


Figure 5.6: These images show an arch before and after smoothing.

the g function in (5.8) with the following

$$g(i,j,k) = \frac{s}{27} \sum_{(l,n,m) \in N(i,j,k)} G[l,m,n]$$

where $N(i,j,k) = \{\{i-1,i,i+1\} \times \{j-1,j,j+1\} \times \{k-1,k,k+1\}\}$ and $s \in [0,1]$ s is a scaling factor. Figure 5.6 shows an example of a part of the volumetric terrain after smoothing has been applied several times at different points.

5.4 *Isosurface Polygonization*

So far, we have ignored how to render the volume although several figures showed rendered volumes. In fact, there are several ways in which we can render volume data. The GPU is remarkably good at direct volume rendering. We can store the volume in graphics card memory and perform ray casting for each pixel to directly render the volume. This is useful, for instance, in medical visualization where we might be interested in several isovalues. However, in the context of volumetric geometry, it seems most useful to consider methods that allow us to convert the isosurface to a triangle mesh which can then be rendered using the regular triangle rendering pipeline.

In the following, we will discuss a class of methods known as isosurface polygonization methods and go into details with a particular of these methods. As mentioned, the zero level isosurface (or set) is the surface we want to extract. The goal of isosurface polygonization is to convert such an isosurface to a polygonal (usually triangle only) mesh.

Perhaps, the most well known algorithm for polygonization of isosurfaces is *Marching Cubes*. In MC, we consider each cell in the voxel grid (whose corners are eight voxels) to be a *polygonization cell*. For every polygonization cell, we find out which corners are inside and which are outside. Thus, for each corner, we have a single bit of information (inside or outside) and together the eight corners provide a byte that we can use as an index to a table that contains a triangle mesh suitable for that precise configuration which separates inside from outside voxels. The vertices of this tiny mesh all lie on the edges of the cell. Using the formula for linear interpolation, we can solve for the position along the edge where the isosurface intersects the edge, and this is where the vertex is placed. The process is illustrated in 2D in Figure 5.7.

Unfortunately, MC has a tendency to produce ugly triangles. This is largely due to the fact that the isosurfaces sometimes intersect the cell very close to a corner and this leads to sliver triangles. Moreover,

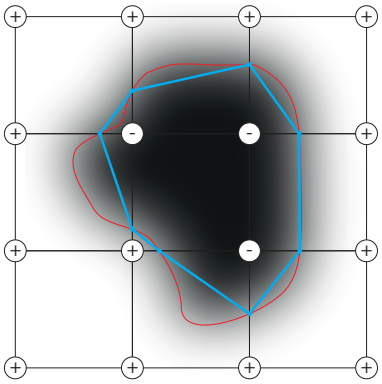


Figure 5.7: Marching Cubes works by approximating the isosurface with piecewise linear geometry inside each polygonization cell. Here the approximation is shown as lines (blue) but in 3D, the geometry is represented as triangles.

while simple, the algorithm relies on a table of mesh templates for all configurations of cells which is simply not necessary using other methods.

5.4.1 *Dual Contouring*

Instead, we shall consider the so called dual contouring (originally surface nets [1]) approach. Compare to MC it does have the drawback that we are not guaranteed a manifold mesh, e.g. some edges might have more than two adjacent faces. Normally this is not an issue in real-time graphics, however, and the result can easily be made manifold if need be. In return for this, the method is exceedingly simple. The solution is to switch from the grid point to the cuboid model of a voxel. Consider each voxel to be a small cube. If its value is below zero (assuming we are after the 0 isovalue) the voxel is inside and if its value is greater than zero, it is outside. Our isosurface representation is simply the set of cube faces which are shared between an inside and an outside cube.

The algorithm is straightforward: Visit each voxel whose value is ≤ 0. From that voxel, visit all six face neighbors. For each neighbor whose value is > 0, we emit a quadrilateral face. The single complication which arises is that the faces are not automatically connected, i.e. if we naively spew faces, each face will have its own separate vertices. A simple solution to this problem is to use a spatial hashing scheme, mapping the position of each vertex to an entry in a hash table which stores the vertex indices. Thus, when we emit a face, we look up all vertices in the hash table. If a vertex is stored in the hash table, we grab its index from there. Otherwise, we produce a new vertex and add it to the array while we add its index to the hash table.

At this point, we have a connected mesh, but it looks like it is composed of sugar cubes. That is not too hard to fix, though. For objects that are implicitly represented (either in voxel grids or as functions on closed form) we often use the gradient as the surface normal, since the gradient is known to be perpendicular to the isosurface of the represented object. The gradient, ∇f of a function f is simply the vector of partial derivatives,

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix} . \tag{5.9}$$

In practice, we compute the gradient using a finite difference scheme. The following code snippet shows our implementation:

[1] Paul W de Bruin, FM Vos, Frits H Post, SF Frisken-Gibson, and Albert M Vossepoel. Improving triangle mesh quality with surfacenets. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2000*, pages 804–813. Springer, 2000

```
Vec3f grad(const RGrid<float>& grid,const Vec3f& p)
{
    Vec3f g(interpolate(grid,p+Vec3f(1,0,0))-interpolate(grid,p-Vec3f(1,0,0)),
            interpolate(grid,p+Vec3f(0,1,0))-interpolate(grid,p-Vec3f(0,1,0)),
            interpolate(grid,p+Vec3f(0,0,1))-interpolate(grid,p-Vec3f(0,0,1)));
    g *= 0.5;
    return g;
}
```

The gradient can be seen as a vector that points in the direction of maximum increase in f. Thus, we can push the node in the gradient direction (or the opposite direction) until it reaches the isosurface. Mathematically, we update a point $\vec{p}$ as follows:

$$\vec{p} \leftarrow \vec{p} - f\frac{\nabla f}{\|\nabla\|^2} \tag{5.10}$$

If f happens to be a distance field the formula above is rather evident. Then the gradient is unit length, so the denominator vanishes. In this case, (5.10) expresses that we go the distance to the isosurface ($|f|$) in the direction of the isosurface ($\pm\nabla f$).

A single update is not enough, unless f is, in fact, a distance field. In practice, we recommend two iterations and also to clamp the maximum distance that a vertex is moved in case there are some ill behaved regions with large gradients. The resulting code is shown below.

```
void push_to_surface(const RGrid<float>& grid, Vec3f& p)
{
    const float tau = 0;
    const float max_dist = 1;
    for(int iter=0;iter<2;++iter) {
        Vec3f g = grad(grid, p);
        double sl = sqr_length(g);
        double d = (interpolate(grid, p)-tau)/sl;
        Vec3f disp = g*d;
        float disp_len = length(disp);
        p -= disp * min(max_dist/disp_len, 1.0f);
    }
}
```

Having pushed the vertices onto the isosurface, we would normally divide the quads into triangles (since the graphics card only renders triangles). The most naive approach is to simply choose a diagonal at random. We recommend the more principled approach of selecting

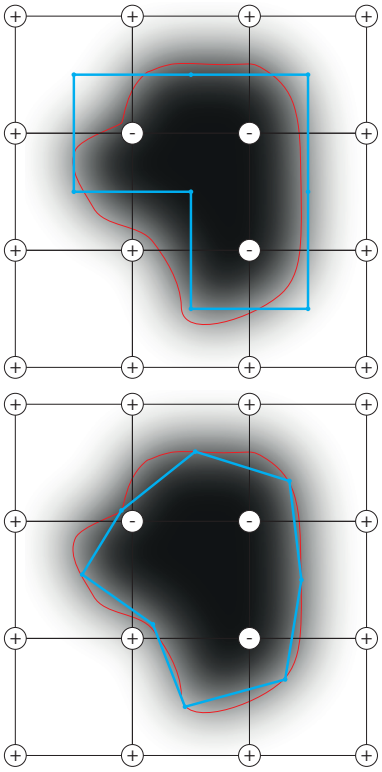


Figure 5.8: A 2D illustration of dual contouring which would yield the result shown on top, i.e. a line segment separating each inside voxel from each neighboring outside voxel. Below is the result after the gradient based procedure for pushing onto the isosurface.

the shortest diagonal. This will usually result in much nicer looking triangles in some places and is not a terribly costly addition to the algorithm. The principles behind dual contouring are shown in Figure 5.8.

The gradient is also highly useful for shading. Since the gradient and the normal are both defined as being perpendicular to the isosurface, they are the same except for normalization. Of course, we can also compute the normals from the triangles, but since the code for estimating gradients is simple and needed for isosurface polygonization, it is arguably better to use the gradients.

5.4.2 Interactivity through Bricking

One concern that has not been addressed so far is the matter of polygonization in the context of interactive editing. For a grid of reasonable size, say $512 \times 512 \times 64$ we would experience a delay if we try to polygonize the entire isosurface every time a small part of the volume changed.

Perhaps the most obvious solution is to not have one big volume but several volumes and then store a triangle mesh for each. Indeed if we want a truly vast voxel world, we will need to think in terms of compressing the volume. Storing the volume as a two level grid, i.e. a top level grid which contains bricks that, in turn, contain the actual voxels is probably the best way to achieve truly vast voxel spaces. The greatest advantage is that if a brick is completely solid or completely empty, we can omit it, leading to a very effective compression since usually the surface does not pass through a majority of the bricks.

Having a two level hierarchy comes at the price of some overhead, though. When we interpolate or compute gradients, we would often need to access voxels belonging to two or more bricks. Consequently, if a single grid is manageable memory-wise then dividing it into bricks will not yield a performance increase, quite the opposite. For instance, a grid with the dimensions above, using a single float per voxel, has a foot print of 64 Mb. In an age where your computer has many GB of RAM, storing such a grid in memory is no longer an issue.

On the other hand, storing the entire triangle mesh produced by isocontouring in one big vertex array is not a good idea since we would then need to update the entire mesh every time the volume changes somewhere. Thus, we recommend to use a grid of triangle meshes. For instance, each entry in the triangle mesh grid might correspond to a region of $32 \times 32 \times 32$ voxels in the volume. If any voxels in that region change, we rerun the isosurface polygonization

only for that region. Assuming we do not make edits that are bigger than the region size, we will need to update at most eight triangle meshes. This is quite acceptable.

5.5 *Learning More*

For many more details on distance fields, the reader is referred to our survey [2]. Of course, in many cases we would like to convert triangle meshes directly to distance fields. A specific algorithm for this is discussed in [3]. We go more in depth with algorithms for isosurface polygonization in our book on geometry processing [4]. Finally, volume graphics and, in particular, GPU based rendering of volumes is discussed by Hadwiger et al. in their book on volume graphics [5].

[2] Mark Jones, J Andreas Baerentzen, and Milos Sramek. 3d distance fields: A survey of techniques and applications. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):581–599, 2006

[3] J Andreas Baerentzen and Henrik Aanaes. Signed distance computation using the angle weighted pseudonormal. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):243–253, 2005

[4] J Andreas Bærentzen, Jens Gravesen, François Anton, and Henrik Aanæs. *Guide to computational geometry processing: foundations, algorithms, and methods*. Springer Science & Business Media, 2012

[5] Markus Hadwiger, Joe M Kniss, Christof Rezk-Salama, Daniel Weiskopf, and Klaus Engel. *Real-time volume graphics*. AK Peters, Ltd., 2006

5.6 *Project: Interactive Volume Graphics*

Objectives

The objective is to exploit the volumetric nature of the scene representation in order to create an editable terrain along with tools for editing.

Part 1

Use a 3D noise function to turn the pure height map terrain into a more interesting cave filled environment. This should be implemented in the `VoxelWorld::build` function. To enable the volumetric environment, you also need to go to `TerrainScene.h` and change it so that `VoxelWorld terrain` is uncommented and `Terrain terrain` is commented out.

Part 2

Implement the smooth and edit functions in the `VoxelWorld` class. Initially, you can simply use the user position as the place where the edit occurs.

Part 3

Consider the user interaction. Create a cursor which indicates where edits happen. Choose a strategy for placing the cursor and design its visual appearance such that it conveys something about what the tool does. This tasks is clearly rather open, and the precise interpretation should be in accordance with your level of ambition.

Deliverables

The code snippets that you wrote yourself along with screenshots that illustrate your results in all three parts.

6
Making Rendering Fast

When we consider how to efficiently render 3D models, it is important to have a good model of how the hardware works and to combine that model with common sense. Such a model along with many of the basic notions of real-time graphics was presented in the introductory chapter, which we encourage you to review before reading the following chapter.

A fundamental premise of real-time graphics is that communication between the host system and the graphics card is a bottleneck and that rendering a single triangle is extremely fast if and only if we are rendering many triangles at a time and these are stored close to the GPU where they are needed. Thus, we need to keep graphics data in large batches as close to the GPU as possible, i.e. in graphics card memory, and we make decisions about rendering a batch of triangles not individual triangles. The first topic of this chapter is how to define those batches and send them to the GPU.

Once we understand the mechanics of that, we need to consider how a triangle mesh should be structured so that we can exploit the fact that most vertices are used more than once. This leads us to indexed meshes and triangle strips which is considered next.

It does not stop there: we very often need to draw the same geometry many times with minor modifications. For instance, we might draw an army or a forest. In this case, we want to draw a solider or a tree many times with at least position and orientation being different from each instance. OpenGL has efficient mechanisms for drawing multiple instances.

We discuss the attractive option of sending compact descriptions of smooth surface patches to the GPU and having the GPU tessellate these into triangles. This has become possible fairly recently and has important benefits when it comes to rendering very detailed geometry.

Methods for exploiting visibility are discussed next: to get the very best performance, it is sometimes necessary to go beyond the

basic tools accorded by OpenGL and consider things like occlusion culling and frustum culling. Another topic is state sorting. This may be important if you are rendering a large heterogeneous collection of objects.

Finally, we close the chapter with some remarks about where you can find more detailed information about issues that were not covered in depth.

6.1 *Sending Triangles to the Graphics Pipeline*

As explained above, we never send a single triangle to the graphics card (using modern OpenGL). The mechanism that we use instead is called *vertex arrays*. A vertex array at the simplest level is simply an array containing the corners of a set of triangles or containing other attributes associated with such corners. It would be nice to introduce that concept gently, but, unfortunately, vertex arrays do not work unless we also use *vertex array objects* and *vertex buffer objects*. A buffer object is a piece of memory that we use for storing data - in this case vertices – and a vertex array object is a mechanism for storing the buffer object state. Furthermore, we need to create a shader program for drawing the triangles. With all this is in place, the shortest possible "Hello Triangle" program is about 60 lines of code. In some sense that is unfortunate. On the other hand the flexibility of the modern, programmable, graphics pipeline allows us to avoid some other complexities that arose using the simpler model of old style OpenGL. The output from Hello Triangle is shown in Figure 6.1.

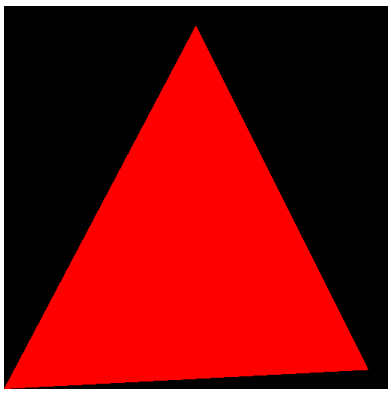


Figure 6.1: Graphics output from Hello Triangle.

To use OpenGL at all, we need to include the header file and also the header file for the library that we use to make the connection between OpenGL and the window system. In this case (and only in this case) we use GLUT since it allows for a very compact example. The include statements, unfortunately, are platform dependent. On a Mac we would use the following:

```
#include <OpenGL/gl3.h>
#include <GLUT/glut.h>
```

Now, we need a function which creates a shader program and returns it. Note that the function below stores the shader `glsl_prog` in a static variable. The shader program is compiled and linked only the first time the function is called. Hereafter, `glsl_prog` is simply returned immediately by the function.

```
const GLuint vertex_attrib_idx = 0;
GLuint get_glsl_prog() {
```

```
    static GLuint glsl_prog = 0;
    if(glsl_prog == 0) {
        const char* vertex_shader_string =
        "#version 150\n"
        "in vec3 vertex;\n"
        "void main(){ gl_Position = vec4(vertex,1.0);}\n";
        const char* fragment_shader_string =
        "#version 150\n"
        "out vec4 fragColor;\n"
        "void main() {fragColor = vec4(1.0, 0.0, 0.0, 1.0 );}\n";
        glsl_prog = glCreateProgram();
        GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
        GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(vertex_shader, 1, &vertex_shader_string, 0);
        glShaderSource(fragment_shader, 1, &fragment_shader_string, 0);
        glCompileShader(vertex_shader);
        glCompileShader(fragment_shader);
        glAttachShader(glsl_prog, vertex_shader);
        glAttachShader(glsl_prog, fragment_shader);
        glBindAttribLocation(glsl_prog, vertex_attrib_idx, "vertex");
        glLinkProgram(glsl_prog);
    }
    return glsl_prog;
}
```

Observe the structure of the code above. We first define some character strings that store the shader program, then we create the program itself, create the shaders, pass the shader source to OpenGL, compile the shaders, attach them to the program, and, finally, we link the program. These operations are all needed for shader program creation and must be carried out in that order. In this book, we will not dwell on all of the parameters needed for each function call. However, you should be clear on what they are. Please peruse `http://www.opengl.org` and find therein the manual pages for the various OpenGL calls.

We need to discuss one more thing: the call to `glBindAttribLocation`. This associates the index, `vertex_attrib_idx`, with the attribute called `vertex`. We can leave the binding of index to attribute to OpenGL, but then we would later have to query the shader program for the index, since we need it when we create the vertex array as shown below.

```
GLuint get_vao() {
    static GLuint vao=0, vbo=0;
    if(vao == 0) {
```

```
        const GLfloat vertices[] = {-1,-1,0, .9,-.9,0, 0,.9,0 };
        glGenVertexArrays(1, &vao);
        glBindVertexArray(vao);
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, 9*sizeof(GLfloat), vertices, GL_STATIC_DRAW);
        glEnableVertexAttribArray(vertex_attrib_idx);
        glVertexAttribPointer(vertex_attrib_idx, 3, GL_FLOAT, GL_FALSE, 0, 0);
    }
    return vao;
}
```

In the code above, we create two objects a vertex array object and a buffer object. First we then put data into the buffer object. Some attention should be accorded to the `glBufferData` call. What this function does is pass data to OpenGL. We always use the bound buffer, hence there is no need to specify the buffer. We just need to give the type of buffer, the size of the data, a pointer and a hint about how we plan to use the data. By specifying `GL_STATIC_DRAW` we indicate that we are not going to change the data but we will be drawing from it. This means that it is best for performance to put the data in the graphics card memory.

Note that all of the calls that have to do with vertex buffers do not affect vertex array object. However, the calls to `glEnableVertexAttribArray` and `glVertexAttribPointer` do. This is where we tell OpenGL which attributes are active (`glEnableVertexAttribArray`) and from where the vertices should be drawn (`glVertexAttribPointer`). The last argument (0) in the call to glVertexAttribPointer is an offset into the bound array buffer. Executing this call stores (in the vertex array object) information about where the data for the attribute `vertex_attrib_idx` should be taken.

Now, rendering is a breeze:

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glUseProgram(get_glsl_prog());
    glBindVertexArray(get_vao());
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glFinish();
}
```

we simply use the created program and bind the vertex array object. Finally, the call to `glDrawArrays` instructs OpenGL to render triangle primitives using the three vertices in the buffer.

For completeness, we also include the few lines of code that al-

low us to use OpenGL at all. Note that you probably should not use GLUT for any serious project, but it is good for getting started quickly on toy programs.

```
int main(int argc, char * argv[]) {
    glutInitDisplayMode(GLUT_3_2_CORE_PROFILE|GLUT_RGBA|GLUT_SINGLE);
    glutInitWindowSize(512, 512);
    glutInit(&argc, argv);
    glutCreateWindow("Hello triangle");
    glutDisplayFunc(display);
    glutMainLoop();
}
```

6.1.1 *Indexed Geometry*

Presently, we ask the question "what if there are more triangles and they share vertices"? The simplest answer to that question is to use triangle strips as discussed in the introductory chapter. To implement triangle strips, we only have to change a few things.

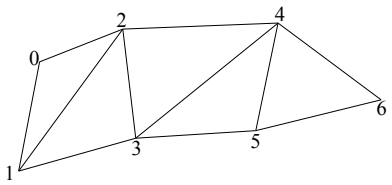


Figure 6.2: A triangle strip is a sequence of vertices where every new vertex (after the first two vertices) gives rise to a triangle formed by itself and the two preceding vertices.

In the function that creates the vertex array, we now have four vertices (and thus 12 floating point numbers). Thus, the code is almost identical:

```
GLuint get_vao() {
    static GLuint vao=0, vbo=0;
    if(vao == 0) {
        const GLfloat vertices[] = {-1,-1,0, .9,-.9,0, -0.85,0.9,0.0, 0.85,.9,0};
        glGenVertexArrays(1, &vao);
        glBindVertexArray(vao);
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, 12*sizeof(GLfloat), vertices, GL_STATIC_DRAW);
        glEnableVertexAttribArray(vertex_attrib_idx);
        glVertexAttribPointer(vertex_attrib_idx, 3, GL_FLOAT, GL_FALSE, 0, 0);
    }
    return vao;
}
```

When we draw, we simply change the called to `glDrawArrays` to

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

to reflect the fact that we are now drawing a triangle strip with four vertices. What happens is that the first three vertices at (-1,-1,0), (0.9,-0.9,0), and (-0.85,0.9,0.0) are used to draw the first triangle. The GPU

then reverses the order of the last two vertices and adds the final vertex to draw the second triangle (-0.85,0.9,0.0), (0.9,-0.9,0), (0.85,0.9,0). The graphics output is shown in Figure 6.3.

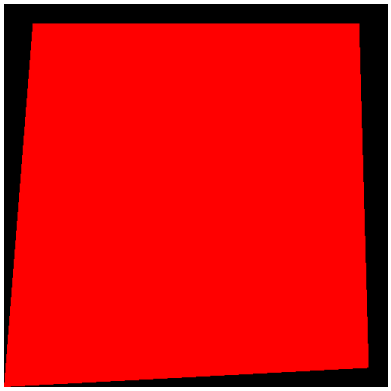


Figure 6.3: Graphics output from Hello Quad.

In general, of course, triangle strips are much longer than just two triangles. In that case, the vertex processing cost of a triangle is reduced to the cost of a single vertex since for each new triangle we just need to send one new vertex which is combined with the previous two vertices. To understand how, imagine that the first vertex is labelled α and the next one β, the third one α again and so on. When a vertex is received, before we label it, we render the triangle formed by the last α vertex, the last β vertex and the new vertex. having drawn the triangle, the new vertex replaces the α vertex if it is odd numbered and the β vertex if it is even numbered.

We often have to resort to multiple strips. The naive way of drawing multiple strips is to make multiple calls to `glDrawArray`. However, there is a better way. In recent versions of OpenGL, we can use a special vertex index as the restart index. There is also a simple solution even if we do not use indices (as indeed we do not in the example). If we simply repeat the first and last vertices in a triangle strip, the strip will contain some degenerate triangles where these vertices are repeated. If we combine two such strips and draw them in one call, the GPU will receive the vertices in the order that looks like this.

```
a1 a1 a2 a3 a4 a4 b1 b1 b2 b3 ...
```

This leads to the triangles

```
(a1a1a2) a2a1a3 a2a3a4 (a4a3a4 a4a4b1 b1a4b1 b1b1b2) b2b1b3 ...
```

where the triangles in parentheses are degenerate (because they contain the same point twice) and not drawn. To sum up, using this method we trade the drawback of some degenerate triangles for the benefit of not having to call `glDrawArrays` multiple times. Performancewise, this will usually be to our advantage.

Triangle strips are convenient for geometry which is very ordered: terrain comes to mind. It seems, though, that graphics programmers have been shifting from triangle strips to indexed geometry. The greatest benefit of strips is that they allow us to reuse the computations performed in the vertex shader. That benefit can also be attained if we simply number the vertices and draw geometry not by passing the points in space but indices to these points' location in the vertex array. Because, if we do so the graphics hardware will store the processed vertices in the so-called *post transform and lighting cache*. When another triangle comes along that needs the same vertex, the processed vertex is taken from the cache.

Drawing our two triangles using indexed primitives, we have to change the `get_vao` function yet again:

```
GLuint get_vao() {
    static GLuint vao=0, vbo=0, ibo=0;
    if(vao == 0) {
        const GLfloat vertices[] = {-1,-1,0, .9,-.9,0, -0.85,0.9,0.0, 0.85,.9,0};
        const GLuint indices[] = {0, 1, 2, 2, 1, 3};
        glGenVertexArrays(1, &vao);
        glBindVertexArray(vao);
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, 12*sizeof(GLfloat), vertices, GL_STATIC_DRAW);
        glEnableVertexAttribArray(vertex_attrib_idx);
        glVertexAttribPointer(vertex_attrib_idx, 3, GL_FLOAT, GL_FALSE, 0, 0);
        glGenBuffers(1, &ibo);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6*sizeof(GLuint),
                                indices, GL_STATIC_DRAW);
    }
    return vao;
}
```

Notice that we are creating an *element array buffer* using the same syntax as when we created the array buffer. This buffer, `indices`,contains indices into the vertex buffer. Thus, we first render the triangle with vertices 012 and then the triangle with vertices 213.

It is confusing, to say the least, that the index buffer is bound to the vertex array object just by binding it with a bound VAO. Recall that binding the array buffer with vertex attributes did not change the VAO. Be that as it may, we can now render the two triangles with a call to `glDrawElements`:

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

In this tiny example, it is not better to use triangles than triangle strips. It does not matter at all. For a complex mesh, things might look different, though. Each vertex is on average shared by six triangles. When we render a triangle strip, we specify a triangle once and use it only three times. Of course, we can render an indexed triangle strip, but there is the remaining issue that, in practice, we may have to deal with many small strips, and there is a cost associated with restarting a triangle strip. For these reasons, it appears that we are moving towards simply using indexed triangles. However, the transform and lighting cache is of limited size (probably around 32 vertices). To ensure that we have as few cache misses as

possible, algorithms have been invented for triangle reordering. The basic idea is to make a simple mathematical model of how a particular or a generic post T&L cache works and then reorder the triangles to ensure the best possible cache performance. Forsyth describes an algorithm that is fairly straight forward [1].

[1] http://home.comcast.net/~tom_forsyth/papers/fast_vert_cache_opt.html

6.1.2 *Multiple Attributes*

This chapter would be incomplete without touching upon the case where we have multiple attributes since we invariably do. Only rarely do we render a mesh where the vertices do not have associated normals, texture coordinates or color or several of these. In the following, we show how to add normals. Again, we change the `get_vao` function

```
GLuint get_vao() {
    static GLuint vao=0, vbo=0, ibo=0;
    if(vao == 0) {
        const GLfloat vertices[] = {-1,-1,0,-3,-0.4,1, .9,-.9,0,-0.3,-.3,1,
                                    -0.85,0.9,0.0,51,0.41,1, 0.85,.9,0,30,0.3,6};
        const GLuint indices[] = {0, 1, 2, 2, 1, 3};
        glGenVertexArrays(1, &vao);
        glBindVertexArray(vao);
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, 24*sizeof(GLfloat), vertices, GL_STATIC_DRAW);
        glEnableVertexAttribArray(vertex_attrib_idx);
        glEnableVertexAttribArray(normal_attrib_idx);
        glVertexAttribPointer(vertex_attrib_idx, 3, GL_FLOAT,
                              GL_FALSE, 6*sizeof(float), 0);
        glVertexAttribPointer(normal_attrib_idx, 3, GL_FLOAT, GL_FALSE,
                              6*sizeof(float), reinterpret_cast<void*>(3*sizeof(float)));
        glGenBuffers(1, &ibo);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                             6*sizeof(GLuint), indices, GL_STATIC_DRAW);
    }
    return vao;
}
```

It is important to note that we do not create a new buffer for the normals but add the normals to the array of vertices. We do this in an interleaved fashion meaning that after three numbers denoting a point in space, we have another three numbers denoting the normal vec-

tor. Thus, we now have a total of 24 floating point numbers (six per vertex). We need to enable a new array index, `normal_attrib_idx = 1`, and call `glVertexAttribPointer` once more for the normals. This time, we also need to specify the stride. The stride is 6 times the size in bytes of a floating point number, since that is how much space we use for the vertex position, normal combination. Clearly, we also need to specify an offset into the buffer since the first normal begins "three floats into" the vertex array.

This could have been done differently. We could have created a different buffer for normals. We could also have stored all the normals after all the vertex positions in the same buffer. Either of these two solutions is likely, however, to be slower than the one presented above. Of course, there could be cases where this does not hold true. For instance, if you need to change one attribute but not the others, it might be a good idea to store them in separate buffers.

6.2 *Instancing and Display Lists*

Now, we can draw a single batch of triangles efficiently, but in many cases, we want to draw the same batch many times but in slightly different ways. Say, we want to draw a forest. In practice it is absolutely unnecessary to have a great many models of trees. We can almost always get away with just a few trees that we reuse since they can be modified with different orientations, scaling, and subtle color changes. One way of expressing this is that we have a state stream (position, orientation, color, scale, etc.) that we merge with a geometry stream.

In fact, this could be done fairly easily using the now deprecated feature known as display lists. Recall from the introductory chapter that display lists are really a macro recording facility for OpenGL commands. What made instancing possible was the fact that display lists can be nested. Thus, the top level display list would contain all of the different state changes and then it would call the bottom level display list which contained the geometry.

However, such reminiscencing is perhaps more pleasurable than useful. Display lists are still available in OpenGL compatibility mode, but that is not what we cover in this book. In current versions of OpenGL the facility we need is simply called *instancing*. Instancing allows us to render a single batch of geometry an arbitrary number of times using a single function call. If we want to render a triangle strip instanced, we replace

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, count);
```

with

```
glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, count, N);
```

The result is that the bound vertex array object is rendered N times instead of just once. Clearly, this is not terribly useful in itself. The advantage is that in the shader, you can access `gl_InstanceID`. If you have uploaded an array of uniforms, `gl_InstanceID` can be used to index into that array. Unfortunately, this is still not very useful. There is only limited storage for uniforms, and you will be limited to a relatively low number of instances: probably hundreds rather than thousands, and you are not likely to see a big performance gain when rendering the same object just a few hundred times.

Fortunately, there are less limiting ways to specify the differences between the states of various instances. Other possibilities include to use a texture to store the state or to use a procedure to compute the state for each instance. However, the most practical solution, which is available from OpenGL 3.3, can be described as a more direct implementation of merging state and geometry streams. The function

```
void glVertexAttribDivisor(GLuint index, GLuing divisor);
```

allows us to specify for the vertex attribute given by `index` how that attribute should be repeated. If `divisor` is zero, the vertex attribute is treated as normally. If the `divisor` is $N > 0$, the attribute is advanced once every N instances. This really does scale to an arbitrary number of instances, but it is limited in two other ways. First of all, it is not possible to pass 4×4 transformation matrices as vertex attributes, and, secondly, the number of attributes that you can pass to a shader is also quite limited. Thus, in practice, to pass a bespoke transformation matrix for each instance of an object, you will need to create four new 4D vector attributes and store these in a vertex buffer. In the shader, these will need to be reassembled into a matrix. Note that this is not a terribly difficult step, but it means that using instances does add complexity to both the CPU bound program and the shader.

Clearly, we should ask whether this complexity is worthwhile? In general, the answer is likely to be "no"! This should not be surprising. Instancing is a feature that addresses the problem of function call overhead, i.e. the overhead of calling a `glDraw*` function N times rather than just a single time. In our experience, even if we are rendering a 100000 instances of an object containing more than 2000 vertices, the function call overhead is so small that using instancing either makes no difference or is even detrimental to performance.

The picture is completely changed, however, if we render a million

instances of an object that contains, say, tens of vertices. In this case, instancing can easily make the difference between real-time and a highly lagging system. Again, this is unsurprising, and it tells us that we should just consider how much geometry is rendered per function call. The less geometry, the bigger the relative gain from instancing, but only for a huge number of instances is the absolute gain going to be significant.

Another thing which is important to take into account is CPU load. If you do use instancing, you will offload the CPU significantly. Thus, while the rendering performance might not increase that much, you are freeing up CPU cycles for other chores which is often an important consideration. If your frame rate is limited by CPU performance, the point at which instancing becomes useful might be far sooner than if the GPU is limiting the frame rate.

To summarize, instancing addresses the problem of function call overhead. This overhead is only significant if we render many objects consisting of few triangles – e.g. a particle system where a mesh is used for each particle – but for those problems, it is an effective means of improving efficiency.

6.3 Geometry Amplification

So far, we have been treating the graphics card as a machine for drawing triangle meshes. It is time to also appreciate the fact that it can transform or synthesize geometry. One way of thinking about this possibility is that we can use the graphics card as a geometry amplifier. We send it a mesh and it outputs a new, far more detailed mesh.

In fact, there are two types of shaders that can be used for such geometry amplification: Geometry shaders and Tessellation shaders. They have a different flavor though. In this text, we will consider geometry shaders in more detail and briefly describe the possibilities of tessellation shaders.

Geometry shaders run after the vertex shader and they have access to a number of vertices. A geometry shader can take as input a point, a line, a triangle, or a triangle with adjacency. The last type of input is confusing. It means that we can send to the geometry shader a triangle plus the last vertex in each of the three triangles that share an edge with the given triangle, i.e. a total of six vertices.

The output from a geometry shader can be points, line strips and triangle strips. The type of output primitive does not have to be the same as the input. For instance, we can input a point and output a triangle strip. This can have some interesting uses. For instance, we can use it to tessellate isosurfaces in volume data.

In the example below, we add a geometry shader to the example from before. Only the `get_glsl_prog` function is shown since the geometry is the same as for the quad with the minor difference that we only send vertex positions and not normals. The geometry shader outputs three smaller triangles (actually three triangle strips with a single triangle each) at the corners of the original triangle. The graphical output is shown in Figure 6.4.

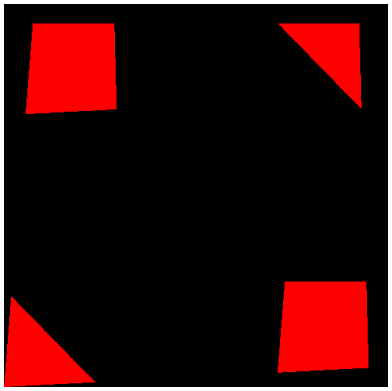


Figure 6.4: Graphics output from Hello Geometry Shader. Each of the two input triangles are transformed into three output triangles.

```
GLuint get_glsl_prog() {
    static GLuint glsl_prog = 0;
    if(glsl_prog == 0) {
        const char* vertex_shader_string =
        "#version 150\n"
        "in vec3 vertex;\n"
        "void main(){\n"
        "  gl_Position = vec4(vertex,1.0);\n"
        "}\n";
        const char* geometry_shader_string =
        "#version 150\n"
        "layout(triangles) in;\n"
        "layout(triangle_strip, max_vertices = 9) out;\n"
        "void main(){\n"
        "  vec4 v0 = gl_in[0].gl_Position;"
        "  vec4 v1 = gl_in[1].gl_Position;"
        "  vec4 v2 = gl_in[2].gl_Position;"
        "  gl_Position = v0; EmitVertex();\n"
        "  gl_Position = 0.75 * v0 + 0.25 * v1; EmitVertex();\n"
        "  gl_Position = 0.75 * v0 + 0.25 * v2; EmitVertex();\n"
        "  EndPrimitive();\n"
        "  gl_Position = v1; EmitVertex();\n"
        "  gl_Position = 0.75 * v1 + 0.25 * v2; EmitVertex();\n"
        "  gl_Position = 0.75 * v1 + 0.25 * v0; EmitVertex();\n"
        "  EndPrimitive();\n"
        "  gl_Position = v2; EmitVertex();\n"
        "  gl_Position = 0.75 * v2 + 0.25 * v0; EmitVertex();\n"
        "  gl_Position = 0.75 * v2 + 0.25 * v1; EmitVertex();\n"
        "  EndPrimitive();\n"
        "}\n";

        const char* fragment_shader_string =
        "#version 150\n"
        "out vec4 fragColor;\n"
        "void main() {fragColor = vec4(1.0, 0.0, 0.0, 1.0 );}\n";
        glsl_prog = glCreateProgram();
```

```
        GLuint vertex_shader = glCreateShader(GL_VERTEX_SHADER);
        GLuint geometry_shader = glCreateShader(GL_GEOMETRY_SHADER);
        GLuint fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(vertex_shader, 1, &vertex_shader_string, 0);
        glShaderSource(geometry_shader, 1, &geometry_shader_string, 0);
        glShaderSource(fragment_shader, 1, &fragment_shader_string, 0);
        glCompileShader(vertex_shader);
        glCompileShader(geometry_shader);
        glCompileShader(fragment_shader);
        glAttachShader(glsl_prog, vertex_shader);
        glAttachShader(glsl_prog, geometry_shader);
        glAttachShader(glsl_prog, fragment_shader);
        glLinkProgram(glsl_prog);
        glBindAttribLocation(glsl_prog, vertex_attrib_idx, "vertex");
    }
    return glsl_prog;
}
```

The geometry shader has many applications some of which are covered in this book. However, one common misconception is that it is useful for subdividing input geometry or tessellation of smooth surfaces. In principle, we can do that, but bear in mind that for a given input primitive, all the output primitives that we produce, are created in a single invocation of the geometry shader. In other words, geometry shaders do not exploit the massive parallelism of current GPUs as well as we would like to. This has changed a bit since OpenGL 4 which allowed for multiple invocations (or instances) of a geometry shader with the same input. However, with OpenGL 4 we also have tessellation shaders. Like the geometry shader, tessellation shaders come after the vertex shader stage (but before a possible geometry shader). Tessellation really consists of three stages:

1. The tessellation control shader (TCS) which determines for each input patch how it is tessellated. This is optional and if no TCS is provided, standard values are used.

2. The tessellator which is a fixed function stage like the rasterizer. The tessellator divides the interior and the edges of a patch into triangles.

3. The tessellation evaluation shader (TES) which computes the positions of the vertices of the subdivided patch.

The fact that computations for each vertex in the output tessellation is computed in a separate invocation of the TES is the explanation for the greater efficiency of using tessellation rather than geometry

shaders for tessellation of smooth surfaces. Observe also that the fixed function tessellator takes a lot of complexity out of the process of tessellation.

In no way should this be taken to mean that the geometry shader has become redundant. On the contrary, we need geometry shaders for a lot of tasks. A good example, covered elsewhere in this book, is wireframe drawing. The single pass method for wireframe drawing requires us to compute for each vertex the distance to the opposite edge of a triangle. That is possible in a geometry shader because it has access to all vertices of the input primitive.

It should be noted that both tessellation and geometry shaders leave us with an issue: the hardware does not retain the geometry after it is rendered. Fortunately, that can be fixed. Using transform feedback, we can stream the geometry back into a vertex buffer object and render it in the usual way. This is often crucial, for instance if we want the detailed geometry to be rendered in more than one pass and the tessellation (or geometry shading) is too computationally expensive to run more than once.

Of course, compute shaders also allow us to write the output to a vertex buffer object. Some developers strongly prefer to amplify geometry in compute shaders, since this is a more generic type of shader which does not have the limitations of geometry shaders or the idiosyncrasies of both geometry and tessellation shaders which they inherit from being a part of the graphics pipeline.

6.4 Exploiting Visibility

So far, we have only considered how to render efficiently using the machinery of OpenGL but not how we can tweak the performance on the CPU side. For simple programs, the answer is that we should not try. As mentioned in the beginning of this chapter, we do not consider individual triangles. OpenGL ensures that invisible triangles are not rendered: triangles outside or intersecting the view frustum are automatically clipped, back facing triangles can optionally be culled and the depth buffer ensures that occluded triangles are removed. Thus, on the triangle level, visibility is already taken care of.

However, in some cases we might want to render a gigantic scene, say, a city model. If the model is too large to render in its entirety, we might indeed want to exploit visibility. Advanced methods for doing so are beyond the scope of this chapter, but let us take a look at some of the simpler options.

In a sense the notion of a level in computer games is a way of dealing with visibility. Presumably, what is outside the level is not visible

(yet) to the user. Moving to the gateway between levels can thus be seen as a poor mans method for dealing with visibility. However, we clearly do not wish to inject levels into our city model.

The next thing that comes to mind is view frustum culling. Whereas it is tricky to know in advance whether a part of the scene is occluded by another part of the scene, it is somewhat easier to find out whether a given part of the scene is inside the viewing frustum. Recall that the viewing frustum is just six planes: the near and far clipping planes and the four planes which form the pyramid whose apex is the eye point (the origin in view coordinates). Assume our scene is divided into appropriately sized chunks. For each chunk, we construct a bounding volume. Here simplicity is the key. We recall the important rule that any fancy processing will probably slow us down and make the test more expensive than the draw operation. What this entails is that a bounding sphere might be a good option. It is easy to check if the sphere lies outside of any one the six planes. If that is the case, we do not draw it. Otherwise, we draw it.

This is clearly a conservative test. There are cases where the sphere is completely outside the frustum but is not rejected by a single plane. There are also cases where the sphere intersects the frustum but all of the geometry inside the sphere is outside the frustum because, well, the bounding volume is not always a tight fit. In any case, this test is very fast and can easily save us from a lot of redundant processing in the OpenGL pipeline.

Occlusion culling is more tricky. If something is terribly expensive to draw and inside the view frustum but not visible because it is covered by something else, how can we know? The graphics hardware does provide some tools to help us with this. It is possible to perform a so called occlusion query, where we send some geometry to OpenGL and are allowed to later query whether anything was drawn. This could be used to draw a bounding volume and query whether that volume is visible before the actual geometry is drawn. For this to work, we clearly need to draw the occluders first and then the occludees. Unfortunately, we may not have a clear division of our scene into occluders and occludees. One way around that is to render the scene front to back. Thus, before rendering an object, we first test whether it is occluded. If not, then it is rendered (and the depth buffer updated) and contributes to the depth buffer. We then take the next closest object and repeat the procedure. While variations of this approach have been shown to work in practice it is not very easy to make it work. A big part of the problem is that we might starve the graphics pipeline because we are waiting for the occlusion queries to be resolved. The method requires many roundtrips to the GPU.

To resolve that issue, conditional rendering was introduced. Es-

sentially, we submit rendering commands which are not immediately executed but contingent on an occlusion query. If the occlusion query (for a cheap bounding object) tells us that the bounding object is visible, OpenGL performs the conditional rendering and otherwise discards it. This saves the roundtrip to the CPU.

In any case, visibility is a challenge at run time. So naturally, if we have a static scene, it is a very good idea to do visibility preprocessing. In its simplest form, this is the method of portals. If our scene is divided into rooms, we can denote these as view cells. Next, we use a ray tracer or real-time rendering to compute the set of objects that are visible from that view cell. This set of objects is often called the potentially visible set (PVS).

Unfortunately, if we simply use rooms as view cells, we might have very suboptimal view cells in some cases. This is easy to understand if you consider a simple case of two rooms. The first room is our view cell. In the other room, which is visible through a portal, we place an object that is incredibly expensive to render. Now, that object could be visible in some places of the first room but not everywhere. In this case, it might lead to a great performance enhancement to split the room into two or more view cells. Thus, if we precompute the potentially visible set for our view cells, it might be best to first precompute the optimal set of view cells.

6.5 State Switching: Avoid That!

In our quest towards more efficient rendering, we have now come to an issue which is important and really just a matter of common sense yet often neglected when teaching real-time graphics and generally hard to deal with.

A sophisticated object oriented designer (who is also a naive graphics programmer) might easily code some framework where a 3D object contains a method by which it can render itself. In fact, the framework that accompanies this book is a bit like that. When a 3D object renders itself, it first loads the shader program and the textures that it needs. It then binds vertex array object and calls `glDrawElements`. Now, the objects are probably stored in a list of such 3D objects. Rendering the scene is just a matter of running through the list and asking each object to render itself. That sounds fine until we realize that if there are thousands of objects we will be switching shaders thousands of times each frame. That is very unfortunate.

There are two approaches to tackling this problem. First of all, we need to combine and merge assets. If objects have very similar shaders, change it so they use the same shader. Combine several

textures into one. It is also possible to have several objects in the same VBO. With these changes in place it may not be necessary to switch a great deal of state when rendering even a long list of objects.

In case, we still need to deal with many state changes, the next thing to do is to actually sort the list of objects that we render so that we change state as few times as possible. If half of our objects use one shader and the other half uses another shader, we need to ensure that we do not switch back and forth between these two shaders but render all of the objects that need one shader first and only then switch shader program.

Of course, this can be a lot of work. Merging shaders, combining textures and sorting by state is something that is often not necessary in simple scenes where we may have only one shader anyway. However, if we are rendering a large scene with very different objects it could easily be worthwhile.

6.6 Learning More

In this chapter, we have covered several topics that relate directly to the OpenGL API. These things are often covered best in material that is available online. For instance, we recommend that you consult the manual (aka man pages) `http://www.opengl.org/sdk/docs/man3/` for the details of OpenGL 3 function calls and `http://www.opengl.org/sdk/docs/man4/` for the details of OpenGL 4 function calls. The OpenGL Wiki `http://www.opengl.org/wiki/Main_Page` is a very good source for clear presentations about how to do things in practice.

The registry `http://www.opengl.org/registry/` contains the actual specification for the OpenGL API, GLSL and historical specs as well. The specification can sometimes clarify things, which are not clear just from reading the man pages, but it is rarely a good place to start.

Clearly, there is also the forum `http://www.opengl.org/discussion_boards` where many knowledgable graphics programmers will answer questions. Not all questions are answered, but it is a very useful resource.

Regarding state sorting and switching, this is discussed in a series of blog posts that are also concerned wit the important topics of state-ful vs state-less APIs and multithreaded rendering: `https://molecularmusings.wordpress.com/2014/11/06/stateless-layered-multi-threaded-rendering-part-1/`

6.7 *Project: Efficient Rendering*

Objectives and Deliverable

The goal of this exercise is to learn a bit about efficient rendering. We will do this by analysing the code for rendering of the tree (c.f. Section **??**) and seeing if we can improve it. For the submission, we require a screen shot of your changed scene, answers to questions and timings with your comments.

Part 1

Instead of a single tree, draw 100 trees. In the appendix you will find code for a function that draws a 100 trees with multiple draw calls. Use this function instead of the tree drawing code in `draw_objects` and make the changes requested in the comments. Note that this function passes a uniform `InstanceMatrix` and `mat_diff` (the diffuse material color - we will skip specular in this exercise) to the shader as uniforms. You will need to create a new vertex fragment shader pair: the vertex shader should now transforms the tree with both `InstanceMatrix` and the model-view-projection matrix (as usual). The material color should be passed from the vertex shader onto the fragment shader and used there instead of the uniform material color. Note that the vertex shader should also transform the normal correctly! Explain how. Time the performance. We suggest using the `QTime` class. Please measure the time it takes to do all the actual work inside the `TerrainScene::PaintGL` function averaged over a number of frames.

Part 2

Change the `truncated_cone` function such that it outputs a single (degenerate) triangle strip instead of a just many triangles. How is it possible to use a single degenerate triangle strip for the entire tree? the vertices and normals are interleaved and stored in the same array. What does interleaving mean? Why is it smart? You also need to change `create_vertex_array_object` to deal with the interleaving. Time the performance and compare to Part 1. Tweak `truncated_cone` so that it produces different numbers of triangles and see whether the difference between triangles and triangle strips becomes more pronounced when the cones are finely tessellated.

Part 3

Change the multiple tree rendering in part 1 so that all trees are rendered with a single call to `glDrawArraysInstanced`. In your shader you now need to change the uniforms `InstanceMatrix` and `mat_diff` to uniform arrays. Time the performance and compare to Part 1 and 2. Comment on timings.

Appendix

```
void draw_trees(ShaderProgramDraw& shader_prog) {
    const int N = 100;
    static GLint count;
    static GLuint vao = make_tree(count);
    static Mat4x4f M[N];
    static Vec4f mat_diff[N];
    static bool was_here = false;
    if(!was_here) {
        was_here = true;
        for(int i=0;i<N;++i) {
            Vec3f p(i%8,(i/8)%8,4); // Set p to some random position.
            M[i] = transpose(translation_Mat4x4f(p)*
                                scaling_Mat4x4f(Vec3f(0.01)));
            // Multiply a random scaling and a rotation on to M[i]
            mat_diff[i] = Vec4f(1,0,0,0); // make this guy random too.
        }
    }
    shader_prog.set_model_matrix(identity_Mat4x4f());
    glBindVertexArray(vao);
    for(int i=0;i<100;++i) {
        glUniformMatrix4fv(shader_prog.get_uniform_location("InstanceMatrix"),
                                    1,GL_FALSE,(const GLfloat*) &M[i]);
        glUniform4fv(shader_prog.get_uniform_location("mat_diff"),1,
                    (const GLfloat*) &mat_diff[i]);
        glDrawArrays(GL_TRIANGLE_STRIP,0, count);
    }
}
```

7
Deferred Shading and Non-Photorealistic Rendering

In this chapter we combine two topics that appear to be completely unrelated but turn out to be strongly connected when it comes to the actual implementation. The first of these topics is deferred shading. The main idea is to defer shading until after all pixels have been written to the frame buffer. That seems contradictory, but if the pixels contain the information needed for shading, we can do the actual shading in one pass over all pixels, and this reorganization can make rendering simpler.

In the first pass, we only have to create some image buffers that can later be used for shading. This can often be done with the same (or nearly the same) shading program for all objects. In the second pass, we compute the lighting using the information about geometry stored in the buffers. A real advantage here is that all pixels are now treated the same. In normal shading, handling N light sources and M materials yields N×M cases that need to be handled separately – either in a complex shader or M different shaders each dealing with the N lights. In deferred shading, we first handle the M differen materials when laying down the buffers, and then we deal with each of the N light sources during shading, so the complexity is only M+N.

The other topic which we cover is non-photorealistic rendering (NPR). The goals of NPR are varied, but the idea is to render things in a certain style for either artistic reasons or because whatever we want to convey with the rendering is more readily appreciated with a non-photorealistic style. It turns out that some NPR techniques, in particular rendering the outlines of objects, are very easy to implement and work well when implemented using deferred shading.

One of this book's authors used to believe that NPR as a whole was just a shortcut to pretty pictures and of no particular interest, since we should do things the right way by rendering objects as realistically as possible. That was plain stupid and this particular author has since repented of this particular view and also of many other

views.

Non-photorealistic rendering is often very far from being a short cut and in many cases it can produce images of very high aesthetic quality (although that is subjective) and it can be used to show things which are hard to see in more "realistic" images. Thus, non-photorealistic rendering is arguably both something that we can do in order to produce an artistic result and something we might do in order to visualize objects with the intent of making good illustrations. Below, we will consider methods for both.

7.1 Rendering to the G-Buffer

When talking about deferred shading, the first notion that we need to introduce is that of the G-buffer, where, ostensibly, the G stands for geometry. A G-buffer can be seen as a screen space scene representation. Instead of rendering color to the frame buffer, we render the position, normal, and material parameters which would otherwise have been used to shade the fragment to an off-screen buffer.

In the framework, which is used in conjunction with this book, we specifically employ a G-buffer consisting of three buffers: a buffer of

- 3D vectors, each containing the position of the triangle fragment rendered to that particular pixel (Figure 7.1) (in eye coordinates),
- 3D vectors, each containing the normal of the corresponding triangle fragment (Figure 7.2), and
- 4D vectors, each containing the material parameters (color and specular coefficient) (Figure 7.3).

Figure 7.1: Visualization of the depth part of the position component in a G-Buffer.

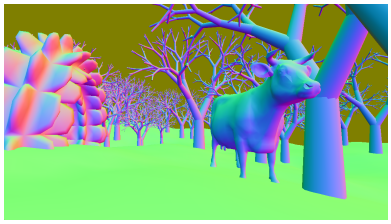


Figure 7.2: Visualization of the normal component in the G-Buffer.

Figure 7.3: Visualization of the material component in the G-Buffer.

This is not the choice that would have been made in a renderer used in the game industry. Normally, the G-buffer does not store the position needed for shading but only the depth value. That is logical as we are rendering to the depth buffer in any case. However, to recover the eye space position which is needed for deferred shading, we must then combine the depth value with the 2D pixel position. Simply storing a 3D eye space position makes the deferred rendering step simpler even if it is less space efficient.

The material representation used is also a bit of a compromise. We choose to store an RGB color which is the color that we use for ambient and diffuse reflectance. The alpha channel contains the coefficient for specular reflection. Thus, we do not store the specular exponent which means that we do not have a parameter that can be

used to set the sharpness of the highlight. We also assume that highlights are white since the color information is not used for highlights. While these are coarse approximations we posit that highlights are, in general, not colored. It would be great to store also the specular exponent, but this way we keep all the material information in a single 4D vector.

It could be argued that we make contradictory choices here. The geometry information is stored explicitly even though it could be recovered from depth and pixel position, while some material information has been omitted in order to use only one buffer for materials. For the purpose of this book, we found these choices convenient, but, again, in a production setting, these choices should be made in a more judicious manner based on the target platform and the type and quality of rendering desired.

7.1.1 Implementing G-Buffer Rendering

With our definition of the G-buffer, we must render to an off screen buffer (and not simply the back buffer). One clear reason why this is necessary is that the regular back buffer does not have enough channels to store both a position, normal, and material for each pixel. Moreover, we want to store normal and positions in 16 bit floating point buffers and visible buffers cannot contain floating point values.

Creating the texture used for the position part of the G-buffer looks as follows:

```
GLuint gtex;
glGenTextures(1, &gtex);
glBindTexture(GL_TEXTURE_RECTANGLE, gtex);
glTexParameterf(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameterf(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA16F, WINDOW_WIDTH, WINDOW_HEIGHT,
                                0, GL_RGBA, GL_FLOAT, 0);
```

where we use the word gtex for this texture since it stores the geometric positions. The ntex stores normals and ctex stores the material colors are created similarly. Typically, we use eight bit RGBA values for ctex.

Having created the three textures, we must create a so called *frame buffer object* in order to render to these textures. Creating the FBO and associating the above defined textures looks as follows

```
glGenFramebuffers(1,&fbo);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
```

```
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_RECTANGLE, gtex, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
                       GL_TEXTURE_RECTANGLE, ntex, 0);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT2,
                       GL_TEXTURE_RECTANGLE, ctex, 0);
glGenRenderbuffers(1,&rb);
glBindRenderbuffer(GL_RENDERBUFFER, rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                      WINDOW_WIDTH, WINDOW_HEIGHT);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, rb);
```

In the code above, we use calls of the form glGenX to create two objects: first the frame buffer object and further down the render buffer used as depth buffer. We also use glBindX calls to first bind the frame buffer object and later the render buffer. Once the frame buffer object is bound, we use the glFramebufferTexture2D to associate each of the textures with the FBO. Then, we create the mentioned render buffer and associate that with the FBO using glFramebufferRenderbuffer.

It may not be immediately clear why the depth buffer is attached to the frame buffer with a different mechanism. The reason is that we can associate two things with an FBO: render buffers and textures. If we later need to read from the rendered-to buffer we must use a texture. However, for our particular G-buffer implementation (which is really not that parsimonious) we do not need the depth buffer after rendering and, therefore, use a render buffer instead. It is possible to read pixels from a render buffer but, generally, we should think of a render buffer as a temporary buffer needed during but not after rendering.

With all that in place, we just need one more notion: multiple render targets. We clearly do not want multiple render passes to draw each component of the G-buffer. Doing it all in one go is perfectly feasible. We set up for the shader to use multiple render targets as follows

```
const GLenum buffers[] = {GL_COLOR_ATTACHMENT0,
                          GL_COLOR_ATTACHMENT1,
                          GL_COLOR_ATTACHMENT2};
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
glDrawBuffers(3, buffers);
glClearColor(0,0,-1000.0,0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

The glBindFramebuffer call informs OpenGL that we wish to render

to the FBO. The glDrawBuffers call ensures that we render to all three components of the G-Buffer. It may not be immediately obvious, why the color is cleared to (0,0,-1000,0). However, this clear color will be written to all pixels. Thus, after rendering we have a position for pixels that were *not* rendered to. This is useful for setting, e.g. background color.

With this in place, we just need a shader. For general objects, the vertex shader looks as follows:

```
#version 150
in vec3 vertex;
in vec3 normal;
in vec3 texcoord;
out vec3 p;
out vec3 n;
out vec3 t;
uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 N;

void main() {
    gl_Position = PVM * vec4(vertex, 1);
    p = (VM * vec4(vertex, 1)).xyz;
    t = texcoord;
    n = N * normal;
}
```

Note that the vertex shader outputs positions p in eye space coordinates. The fragment shader follows below

```
#version 150
uniform vec4 mat_diff;
uniform vec4 mat_spec;
uniform sample$r_2$D tex;
in vec3 p;
in vec3 n;
in vec3 t;
out vec4 fragData[3];
void main() {
    fragData[0].rgb = p;
    fragData[1].rgb = normalize(n);
    fragData[2] = mat_diff*texture(tex, t.xy);
    fragData[2].w = length(mat_spec.xyz);
}
```

Notice that our output, fragData, is now a vector of 3 4D vectors – and of course each will go to its appropriate color attachment which has an appropriately bound texture. In the code framework, we have encapsulated the G-Buffer rendering in the GBuffer class. In order to use the contents of the G-Buffer, we need to bind the three created textures, and then we can obtain position, normal, and material information from these shaders during rendering. Binding the textures is also handled by the GBuffer class.

7.1.2 *Shading with the G-Buffer*

In order to actually use the G-Buffer, one would typically render the light geometry. In other words, if a small point light is rendered, we could draw a spherical volume enclosing the region that it affects. Then shading is computed for that light source for precisely the pixels that it influences. However, in the following, we will assume that there is just a single light source affecting the entire scene. In this case, we can clearly just render a screen filling quadrilateral. Arguably, using a single light only, we do not fully exploit the potential of deferred rendering, but, as we shall see, there are several advantages to deferred rendering, even in the case of just one light.

Drawing a single quad is a simple task (covered in Chapter 6) and doing so will cause our fragment shader to execute for each pixel on the screen. We can now make use of the G-Buffer. In the GLSL language, specifically in a fragment shader, we have access to gl_FragCoord which is a variable that contains the x, y, and $1/z$ coordinates of the fragment. We have specified that the texture is a GL_TEXTURE_RECTANGLE, and we have created the G-buffer to have a one to one correspondence with the on-screen window. For such a texture, making a texture lookup at gl_FragCoord.xy gives us the texel that precisely corresponds to the pixel we are currently shading. Thus, we can look up the position, normal, and material parameters and output a shaded color. A dumbed down example which simply outputs the material color, or blue for pixels not found in the G-Buffer, is shown below. If we change it to unconditionally output material color, we would obtain precisely the result in Figure 7.3.

```
#version 150
uniform sampler2DRect gtex;
uniform sampler2DRect ntex;
uniform sampler2DRect ctex;
out vec4 fragColor;
void main()
{
```

```
        float z = texture(gtex, gl_FragCoord.xy).z;
        vec4 base_color = texture(ctex, gl_FragCoord.xy);
        if(z == -1000.0)
                fragColor  = vec4(0.4,0.3,1,0);
        else {
            fragColor = base_color;
        }
}
```

7.1.3 *Advantages and Disadvantages of Deferred Shading*

From a performance point of view, it is clear that when rendering deferred, we shade only pixels that do appear in the final image. Of course, that is also true if we use a simple depth pre–pass where we first render to the depth buffer and then render all geometry again, shading only fragments with a depth match. Going fully deferred, though, we also avoid a combinatorial explosion in the complexity of our shaders, because lighting calculations are now decoupled from materials. As was mentioned in the introduction, the combination of lights and materials can lead to either complex or a great multitude of shaders. Furthermore, if we have light sources that cover only part of the image, we also avoid culling those lights in the shaders, since we can use the light geometry to determine the pixels it affects.

From a different point of view, the G-buffer is a simplified, image space representation of our scene and its geometry. This makes some things very straight forward. A good example is the drawing of the silhouette edge of the object. Another example is screen space ambient occlusion which will be covered in a later chapter.

On the other hand, deferred rendering does introduce one more pass and – for visible pixels – it is not clear that we do less work. Thus, the performance advantage of deferred rendering will typically only materialize when the lighting computations become very heavy.

Other issues are that anti-aliasing and transparency are really hard to do well in conjunction with deferred shading. Thus, we should probably use traditional forward rendering when we can, and deferred rendering when we must.

7.2 *Drawing Edges*

A very important part of NPR is the rendering of edges. By "edges" - in this particular context - we first of all understand the silhouette or contour curves of the object that are often traced with a black line in actual cartoons. It seems that tracing the edges of an object makes it

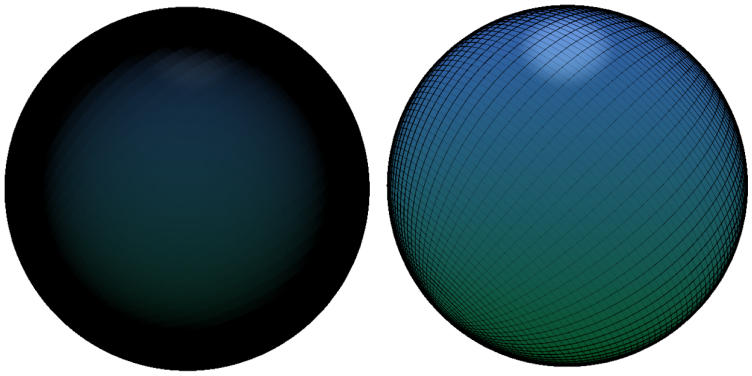


Figure 7.4: The difference between using depth gradient (left) and depth Laplacian (right)

easier to read a drawing. However, it is not only the curves that make up the silhouette that we want to trace. If there is a discontinuity in the depth field within an object or a normal discontinuity, we also want to trace that edge. Often discontinuities between different materials are also traced. In short, it seems that any sort of discontinuity is generally highlighted in toon style rendering although, of course, the artist should be the judge of precisely which edges to highlight.

There are several ways in which we can draw edges, but in many ways the easiest approach is to do it in image space. This basically means a two step procedure where the first step is to render the scene to a G-buffer and to then compute edges from the G-Buffer. So, how do we detect edges in the G-Buffer?

We are really looking for discontinuities. To be concrete let us look for discontinuities in the depth buffer. Our first guess might be to look at the depth difference between adjacent pixels. In practice, we would like to look at the differences between the current pixel and the neighbors in both directions along both the x and y directions. Quickly, this leads to

$$G[x,y] = \sqrt{(z[x+1,y] - z[x-1,y])^2 + (z[x,y+1] - z[x,y-1])^2} \ ,$$

which is recognized by some as the length of the gradient approximated using central differences. If this gradient is big, we probably have an edge. A viable solution is to simply threshold G but we have to take care. A big depth gradient does not necessarily mean that there is an edge there could also be a steep slope as illustrated in Figure 7.4 (left) where we simply fade to black for big values of G. A more judicious function to use is the following

$$L[x,y] = |z[x+1,y] + z[x-1,y] + z[x,y+1] + z[x,y-1] - 4z[x,y]| \ ,$$

which is recognized as the absolute value of the Laplacian. In other words, from a mathematical point of view, L is (an approximation of)

the absolute value of the sum of second order derivatives of z. The Laplacian has the virtue of being zero on linear slopes. This is shown in Figure 7.4 (right). We observe that using a threshold on L picks out the edges between faces but does not generally darken the slope.

In principle, we can apply L to any signal by replacing z with some other value defined per pixel. For instance, we can apply L to the normals in the normal component of the G-Buffer. Of course, the normal is a vector and not a scalar like the depth, but we can apply the function to each coordinate independently and then take the length of the resulting vector instead of the absolute value. Figure 7.5 shows the result of using L together with both depth and normals and also shows the result of combining the two types of edges.

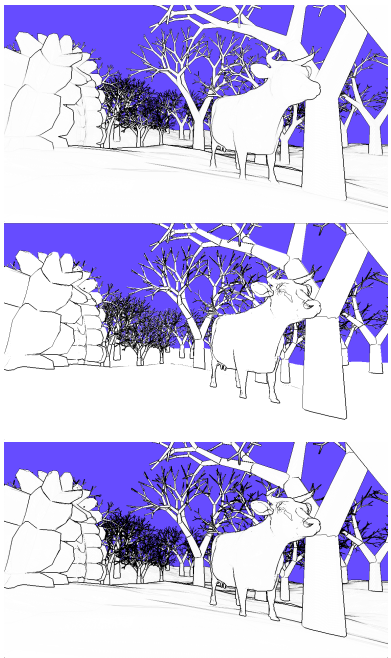


Figure 7.5: From top to bottom, this image shows the edges computed from a G-Buffer by applying a threshold on L to (top) the depth at each pixel, and (middle) the normal at each pixel and finally (bottom) the combination of depth and normal edges.

7.3 *Non-Photorealistic Shading*

Having discussed how to draw the outlines of objects and their parts, we now discuss several strategies for non-photorealistic shading of the interior of a rendered object. Many of these methods could be combined with edge drawing to achieve a certain look. Note also, that the methods below mostly work well both when rendering deferred and when rendering in the normal way.

7.3.1 *Toon Shading*

Modern comic books are often very sophisticated when it comes to coloring the pictures, and smoothly graded colors are probably the norm these days. However, the classical cartoon look was more flat with large areas of the picture having the exact same tone. Probably, that facilitated printing on paper. It also gives a clean and simple look that can be quite impressive. In Figure 7.6 a toon shaded eagle is shown. Just three colors are used: black, dark magenta (brownish), and red. In addition edges have been drawn using an image space technique as discussed previously and there is also a faint Phong highlight.

Figure 7.6: Toon shaded eagle model. Depth and normal edge filters were also applied.

Toon shading is one of those things that are easy to do, but probably hard to do really well. In its simplest form, toon shading is simply a quantization of the diffuse part of Phong shading (also known as Lambertian shading). Thus, we divide the intensity of diffuse shading into a number (two or three) of bands and quantize the intensity within each band to the average intensity of that band. Thus, for all pixels whose intensity lie within a given intensity band, we get precisely the same tone or color in the output. It is worth pointing out that the curves which delimit areas with a different tone are curves where the intensity lies precisely on the boundary between two intensity bands. Thus, they would have the same diffuse shading and hence the same angle between normal and light source direction. Such curves are called *isophotes* meaning that they receive the same illumination. Figure 7.7 shows the combination of toon shading and tracing of contour edges and normal discontinuities.

Figure 7.7: The combination of toon shading, silhouette edges, and normal edges is shown in the picture above.

What do we do in the presence of more than one light source? The answer is that we first accumulate the light contributions and then quantize the sum of the light contributions rather than first quantizing and then adding. Quantizing and then adding would clearly negate much of the simplification that toon shading provides – besides it looks terrible in actual practice.

7.3.2 *Gooch Shading*

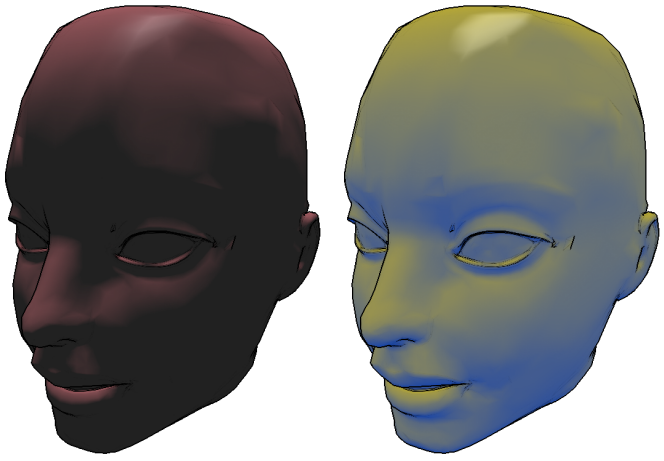


Figure 7.8: A head shown with Phong shading (left) and Gooch shading (right).

While toon shading arguably reduces visual information, Gooch shading[1] does quite the opposite. The important observation when it comes to Gooch shading is that Phong shading does very little to

[1] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 447–452, New York, NY, USA, 1998. ACM

help us understand the shape in regions that are in shadow. This is shown in Figure 7.8. The head model is lit from above, and, using Phong shading, we cannot see any of the features which are in shadow. Of course, we could simply add a second light source, and Gooch shading can be seen as a slightly more principled take on that. Given a direction towards the light source, $\vec{l}$ and a normal, $\vec{n}$, we compute the intensity,

$$I = \mathrm{k}_{\mathrm{warm}} \frac{1 + \vec{l} \cdot \vec{n}}{2} + \mathrm{k}_{\mathrm{cold}} \frac{1 - \vec{l} \cdot \vec{n}}{2} ,$$

where $\mathrm{k}_{\mathrm{warm}}$ and $\mathrm{k}_{\mathrm{cold}}$ are typically yellowish and blueish colors, respectively, which are easily differentiated. Clearly,the formula just interpolates between these two colors using $\vec{l} \cdot \vec{n}$ as the interpolation parameter.

As also noted by Gooch et al. we can plug the original color into the above equation as $\mathrm{k}_{\mathrm{warm}}$ and use black as $\mathrm{k}_{\mathrm{cold}}$. This will look much like Phong shading, but the color will depend on the normal direction also when the normal points away from the light source. In practice, Gooch et al. propose a combination where $\mathrm{k}_{\mathrm{warm}}$ is the weighted sum of a warm color and the original color and $\mathrm{k}_{\mathrm{cold}}$ is the weighted sum of the cold color and black.

7.3.3 *Strokes*

In some cases, we wish to reproduce the effect of an artist's hatchings. Very briefly described, hatchings are closely spaced lines that give an impression of a tone. The thickness of each stroke as well as the precise spacing can be varied to produce different intensities. One reason for rendering objects hatched is to give an impression of an object drawn with a pencil or a pen. However, if we see the direction of the hatches as the direction of a vector, we can also see this method as a technique for rendering vector fields on surfaces. Figure 7.9 shows a simplified version of the Stanford Bunny rendered in this way.

Figure 7.9: The direction of minimum curvature visualized using a hatch like shader.

Many techniques used for this kind of rendering require a 2D parametrization of the object. However, the figure was created in a simpler fashion. For each vertex, we compute the minimum curvature direction (although any vector field could be used) and then we use a 3D noise texture for the strokes. Very briefly put, the fragment shader computes the integral of the noise function along a short line segment in the direction of the curvature direction. This has the effect of blurring noise in the direction of the vector, producing the hatches shown.

7.4 Learning More

Clearly, this short chapter has only broached the subject of non-photorealistic rendering. Methods for computing edges (outlines, silhouettes, etc.) are described well in Real-Time Rendering[2]. For a more general introduction to non-photorealistic rendering, there are several books, but Markosian's PhD thesis is also a good place to start [3].

[2] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 2008

[3] Lee Markosian. *Art-based Modeling and Rendering for Computer Graphics*. PhD thesis, Providence, RI, USA, 2000. AAI9987803

7.5 Project: Toon Rendering

Objectives

Deferred shading is a technique for rendering 3D models where the shading is computed on a per pixel basis after rasterization of all triangles. This means that certain geometric attributes, in particular the normal, need to be stored for every pixel in the image in a buffer sometimes referred to as a G-buffer. Toon shading is simply non-photorealistic rendering in a style that resembles a cartoon. The objective of this exercise is to do toon shading in a deferred fashion.

Practicalities

This project is a part of the TerrainScene. The code you write should be entered in the fragment shader called `deferred_toon.frag`. In this file, the uniform samplers `gtex`, `ntex`, and `ctex` represent the geometry, normal, and color parts of your G-buffer. `TerrainScene.cpp` should be consulted for reference.

Part 1

In `render_deferred_toon()` the rendering is performed in two passes: in the first pass we use multiple render targets to simultaneously output to three buffers (collectively called the G-Buffer as described in this chapter): A geometry buffer containing the 3D position of each pixel (x,y, and z) in eye space coordinates, a normal buffer, and a color buffer containing the (unshaded) color of the object in the rgb channels and its specularity (assuming specular reflection is white) in the alpha channel.

In the second pass, these buffers are bound as textures and a screen filling quadrilateral is drawn with a shader program that simply draws the unshaded color.

Change the program to compute a simple Phong shading with a diffuse contribution, ambient contribution, and specular highlight. You will not be able to obtain exactly the same result as in direct shading from the first exercise due to the fact that in the color buffer, the specular color is only represented by an average in the alpha channel. Now, you have implemented deferred shading. Compare the performance to the normal rendering you implemented in the first exercise. Which is faster? In terms of performance, what advantages does deferred rendering have over normal forward rendering?

Part 2

In this part, you implement toon shading. Change the shading program so that the color from the color part of the G-buffer is modulated not smoothly but simply using a step function applied to the dot product of normal and light source direction. Add a highlight which again should not be graded - if there is highlight, you add a constant term.

Part 3

Add dark strokes around the outlines and along sharp creases of the objects. These strokes should be computed in an image based fashion in the same shader as the toon shading and again from the contents of the G-buffer. We would like you to draw dark lines at both silhouettes and sharp edges. Silhouettes are defined as discontinuities in the depth buffer, and sharp edges are defined as discontinuities in the normal buffer. The final image produced by your program (Parts 2 and 3) should be similar to the image in Figure 7.7, but not necessarily identical. For instance, it is not mandatory to include the trees in this exercise.

Describe your approach to finding the above mentioned discontinuities. Apart from the image based approach taken here, what other options are there for detecting and drawing silhouettes and sharp edges? Discuss the advantages and disadvantages of the methods you describe and of this one.

Deliverables

Answer all questions, hand in shader code (not C++ code). Screenshots of all parts.

8
Shadows and Ambient Occlusion

To people with little knowledge of the machinery behind real-time graphics it often seems surprising that real-time graphics – in its basic form – does not consider any sort of indirect illumination beyond the ambient term, which is based on the assumption that all points receive precisely the same amount of indirect illumination. For many types of scenes, however, direct illumination is sufficient as long as we take *cast shadows* into account. Without shadows, scenes are often hard to read. Consider, for instance, an object placed somewhere in a fairly featureless scene. If the object does not cast a shadow it is impossible to tell if it is standing on the ground or floating in the air. One of the most used techniques for shadow casting is shadow maps which we discuss in this chapter.

As noted in the chapter on non-photorealistic rendering, Phong shading is completely featureless for surfaces pointing away from the light source. Clearly, we can add more light sources, but that may not be appropriate for the scene. What is needed is a method for making surfaces – which are not lit – a bit less featureless. That is precisely what *ambient occlusion* does. It is simply an improvement to the widely used ambient term. We can see the ambient term in Phong shading as the crudest conceivable model for light incident on a surface which comes not directly from the light source but is reflected off of other objects in the scene. Clearly, even if we assume that ambient light is roughly the same in all areas and all directions, some parts of the scene (corners, holes, and crevices, for instance) will receive less of the ambient light due to occlusions. Hence, ambient occlusion was invented, and, in this chapter, we consider the family of techniques collectively called screen space ambient occlusion which are generally easy to implement in the context of deferred shading.

Figure 8.1: The same scene shown with only Phong shading (top) and Phong shading combined with shadows and ambient occlusion (below). While artifacts are visible, it is clear that these two methods greatly enhance the visual appearance of the scene.

8.1 Shadow Mapping

We are used to thinking of texture mapping as a way to glue a 2D image onto a 3D model, but this is just one of many ways in which we can use textures. *Projective texture mapping* is a very different technique where we project the texture onto our 3D model. This is most easy to picture by imagining that the texture is projected from, say, a data projector onto the scene. In practice, what happens is the reverse. Given a fragment that we wish to texture, we project it back into the texture – using exactly the same process as when we render an image.

The idea behind a shadow map is to leverage projective texture mapping for the purpose of rendering cast shadows. First, we render the scene as seen from the light source. This rendering is our representation of the shadow and when we render the scene from the normal viewpoint, we can find out whether a given fragment is in shadow by reprojecting it into the image generated from the light source.

The projection created from the light must be matched to the type of light source that we are dealing with and the area of the scene that it covers.

Figure 8.2: A teapot and an oval shape and the view frustums for the eye and the light source.

8.1.1 Casters and Receivers

In order to understand shadow mapping, we also need to consider *how* we render the scene seen from the light. A simple way of using projective texture mapping to do shadows is to divide the scene into shadow casters and shadow receivers. The only objects that we render are the shadow casters. this provides us with a black and white image - black where we see the shadow casters. We now use the rendered image to texture the shadow receivers. Wherever a point on the receiver projects onto a point in the shadow image that is covered by the image of a shadow caster, we consider the point in shadow and otherwise illuminated with respect to the light source used to generate the shadow map. This clearly requires that the same projection is used to both draw the casters and to check the shadow map for the receivers. In Figure 8.2, we would probably consider the teapot to be the caster and the oval shape to be the receiver.

It is slightly confusing that we have two model-view-projection transformations going on. Clearly, there is the model-view-projection transformation used to render the scene and then there is the model-view-projection used to transform a point into *light space* as we call the image drawn from the light source.

The shadow test is performed in the fragment shader. Often the

tricky part when it comes to shadow mapping is the coordinate conversion needed to go to light space. We will assume that we know the eye space coordinates of the fragment with respect to the camera used for rendering (i.e. not the light source). The transformation needed is then to go from the eye space of the camera back into world space (using the inverse of the view transform, $\vec{V}_e^{-1}$) and then perform the view transform needed for the light source, $\vec{V}_l$, the projection needed for the light source, $\vec{P}_l$, and finally the scaling, $\vec{S}_{0.5}$, and translation, $\vec{T}_{0.5,0.5}$, needed to get the right texture coordinates. All told, we need this transformation

$$\vec{M} = \vec{T}_{0.5,0.5}\vec{S}_{0.5}\vec{P}_l\vec{V}_l\vec{V}_e^{-1}$$

where $\vec{T}_{0.5,0.5}$ is a 2D translation by 0.5,0.5 and $\vec{S}_{0.5}$ is a uniform scaling by 0.5. We have made the simplifying assumption above that the projection is orthographic since for a perspective projection, we would need to divide by w before applying the translation and scaling.

8.1.2 Using a Depth Map

Drawing shadows as described above has the strong limitation that we need to divide objects into shadow casters and receivers, and, in general, the same objects should be both casters and receivers. In fact, an object can cast shadow onto itself.

Fortunately, this is quite feasible. Instead of merely drawing a black and white shadow image, we draw a depth map (still seen from the light source). Now, we test not whether a fragment projects to a point inside a shadow region but whether the depth of the projected position of the fragment is smaller than the depth of the point stored in the shadow map. If the depth is smaller or equal then the point is not in shadow. If the depth is greater, the point is in shadow.

The astute reader will have spotted a small problem. If we no longer distinguish between shadow casters and receivers, then any fragment that is not in shadow will also have been visible when rendering from the light source, so the depth value can never be smaller than whatever is stored in the light texture. If the depth is equal, the point is not in shadow. If it is greater, the point is in shadow. Unfortunately, the depth buffer is not precise, and, moreover, it is increasingly imprecise the farther we get from the near plane as shown in Figure 2.8. Referring to that figure, the perspective projection can be seen as a transformation that maps a pyramid (with the top cut off by the near plane and the bottom by the far plane) to a cube. This transformation preserves straight lines, but warps space inside the cube such that things near the far plane are greatly compressed. The

closer the near plane is to 0, the greater the distortion. Thus, to get the best shadow precision, we need to carefully push the near plane as far away from the eye as possible. Potentially, we could also compute and store the true depth value for each fragment. Note, though, that modifying fragment depth tends to invalidate early z culling.

In any case, the depth buffer precision is always limited, and to avoid a lit surface from shadowing itself, we need to add a small bias to the depth map before comparing the depth as seen from the light source in the current fragment with the depth that was stored when rendering the shadow map from the light source. Actually, there is one other possibility: we can omit rendering the front facing surfaces, but this also leads to problems in cases where the surfaces are single sided or very thing.

When rendering to a shadow map, we need a throwaway render buffer to store the actual pixels and a depth buffer which we can later use for depth testing. Both must be bound to a frame buffer object buffer we can render to the shadow map. Afterwards, when we apply the texture, the depth map is bound as a texture. The combined setup procedure for both the texture and FBO is shown below:

```
glGenTextures(1, &dtex);
glBindTexture(GL_TEXTURE_2D, dtex);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LESS);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, dim, dim, 0, GL_DEPTH_COMPONENT,
GL_UNSIGNED_INT, 0);

glGenFramebuffers(1,&fbo);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);
glGenRenderbuffers(1,&rb);
glBindRenderbuffer(GL_RENDERBUFFER, rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, dim, dim);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER, rb);
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, dtex, 0);
```

8.1.3 Changes to the Shading Procedure

So far, we have not discussed precisely how the shadows affect the lighting. The answer is simple: the shadow should only affect the light that is deemed to come directly from the light source. In other

words, the diffuse illumination and the specular highlight are affected. The ambient term, though, is unaffected, and if we compute specular reflections of the scene, these are also unaffected.

Unfortunately, a few more opportunities for getting unsatisfying results are strewn in our way. From a point light source or a directional light source, it is clearly a binary decision whether a single point is in shadow. Unfortunately, the shadow map has limited resolution. Thus, if we simply find the texel in the shadow map that contains a given point and perform the depth test in this one texel, we will be able to quite clearly see the texels of our shadow map. Visually, it is much better that the edge of the shadow is a little bit fuzzy. This is even more true because no light sources we meet in practice are truly point sources or directional sources.

The solution is to use texture interpolation, but we need to do it differently. We could interpolate the depths stored in a shadow map and then compare against the interpolated depth, but the interpolated depth does not correspond to any true geometry. The more correct way of doing it is to first perform the depth comparison and to then interpolate the result. This is called Percentage Closer Filtering (PCF). Fortunately, if our texture sampler is a Sampler$_2$DShadow the graphics hardware will automatically perform the depth test and then interpolate the results. Thus, PCF is supported in hardware although only for four shadow samples.

A nitty gritty detail that is often forgotten is that the shadow map is only defined inside a finite domain (often the uv space $[0,1] \times [0,1]$. Thus, we need to test whether the point projects to the interior of this domain. If it does not, we should generally assume that the point is not in shadow.

Several different types of artifacts tend to appear when we apply shadow maps. First of all, the shadows look blocky (or aliased). This is hard to get rid of even with a high resolution shadow map. Probably, the best solution is to have several shadow maps - one for the objects that are close to the eye point, one for the middle distance and one for far away objects. While that works it requires us to draw three shadow maps. An idea that was proposed by Stamminger and Drettakis[1] involved warping the shadow volume to ensure that the resolution would match the on screen pixels, but in practice this is hard to make work.

[1] Marc Stamminger and George Drettakis. Perspective shadow maps. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 557–562. ACM, 2002

Other issues arise due to the biasing. If we apply too little, we get shadow stripes in regions that should be fully lit. On the other hand, with too much bias, we get "Peter Paning" where objects appear to mysteriously float above the surface that receives their shadow. .

8.1.4 Penumbra Regions

For a light source that is neither directional nor a point light we would generally also use a perspective projection, but this is where shadow maps become an approximation. For a point light or a directional light, the shadow is, in principle, perfectly sharp: any point in the scene is either in the shadowed (umbra) region or not. For an area light source, however, a point can also be partly lit if it is in a region where the light source is partly visible. Such regions are called *penumbra* regions, and we use the term soft shadows for the corresponding shadows. There are practical methods for computing soft shadows, but it is either much more computationally demanding or considerably harder to implement than hard shadows. Moreover, none of these methods are correct since we cannot capture an accurate penumbra by rendering from just one light source. The correct – but very computationally demanding – solution is to render shadow maps from many points on the area light source.

8.2 Skylight and Ambient Occlusion

In outdoor environments, shadows are often a combination of nearly hard shadows due to direct sunlight and very soft shadows due to skylight. As mentioned before, we can approximate sunlight by a directional light source. The very soft shadows due to skylight are however more appropriately obtained by having light coming in from all directions. In principle, we need low intensity directional sources from all directions. This is sometimes referred to as an *infinite area light*: an infinitely faraway area light source that surrounds the entire scene. Suppose we use an infinite area light of unit intensity. The contribution from such a source is referred to as *ambient occlusion*. Full contribution means no occlusion. Thus, standing in a surface point and considering the hemisphere above this point, the ambient occlusion value is the fraction of the infinite area light in this hemisphere which is not shadowed by scene geometry.

One of the first approaches to GPU accelerated ambient occlusion was to simply use 128 to 1024 directional lights distributed uniformly across the sphere (or hemisphere if there is a ground plane).[2] Since the infinite area light is of unit intensity, the incident illumination from each direction is in effect the visibility term v, which is 1 if the directional light is visible and 0 otherwise. The light diffusely reflected from a surface point $\vec{p}$ is then the following cosine-weighted

[2] Matt Pharr and Simon Green. Ambient occlusion. In *GPU Gems*, chapter 17. Addison-Wesley, 2004

integral of the visibility term over the hemisphere Ω above $\vec{p}$:

$$\frac{\rho_d}{\pi} \int_\Omega v(\vec{p}, \boldsymbol{\omega}) \vec{\omega} \cdot \vec{n} \, d\omega \tag{8.1}$$

where $\vec{n}$ is the normal at $\vec{p}$, $\vec{\omega}$ is the direction toward the incoming light, and $d\omega$ is an element of solid angle. The arguments of the visibility term state that we are considering the visibility at $\vec{p}$ in the direction $\vec{\omega}$. The constant ρ_d is the diffuse reflectance of the material (comparing to the Phong illumination model, we have $k_d = \rho_d / \pi$).

Note that skylight resembles the light received from the surroundings in a diffusely lit environment. Another use of (what we call) ambient occlusion is therefore to approximate the *indirect illumination*. In this case, we would not want to use an infinitely faraway area light, as a point in an indoor environment would be entirely occluded. To approximate indirect illumination, we might exchange v for a function of the distance to the first intersection point of a ray from $\vec{p}$ in direction ω. To be more concrete, we could use

$$v(\vec{p}, \vec{\omega}) = \min \left(1, \left(\frac{t}{R} \right)^2 \right),$$

where R is the maximum distance at which we would check for occlusions and t is the distance to the nearest occluder. In this case, we simply assume that there is some ambient light in the scene such that any point receives the same amount of ambient light – and receives the same amount from all directions in the hemisphere which are not blocked by an occluder. For occluded directions, the function v gives us a scaling factor for the incident ambient light.

In terms of the Phong illumination model, we can now multiply the computed ambient occlusion term from (8.1) onto the ambient light contribution.

8.3 Screen Space Ambient Occlusion

Unfortunately, there is no simple method to compute (8.1). Clearly, one possibility is to compute the ambient occlusion in a preprocess and bake the result into a texture or store it at vertices. However, developers at CryTek recognized the value of computing AO directly during rendering.[3] If we do so, there is no AO assets that need to be updated before the scene can be used and it clearly also works better in the presence of dynamic objects.

The CryTek method which is illustrated in Figure 8.3 – as well as all other SSAO methods – requires a G-Buffer which must contain

[3] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007

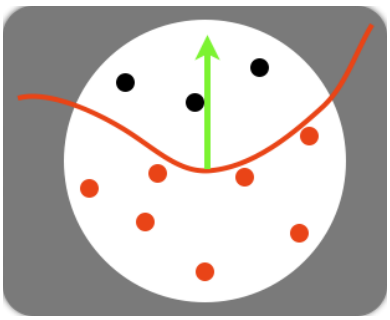


Figure 8.3: An illustration of the scheme

at least a depth buffer. Thus, in the following, the reader is asked to picture a depth buffer rendered from the point of view of the camera and not not light source. We stress this since the shadow map is also a depth buffer but rendered from the light source. In the following, however, we only refer the *G-Buffer depth buffer* which is rendered from the camera.

With this in place, the CryTek method works as follows: given a fragment, we look up the corresponding point $\vec{p}$ in the depth buffer. In the context of this book we have an easier job since the G-Buffer contains not only depth but the full eye space position of the pixel, but even if we only have depth, we can transform pixel position + depth to an eye space position. Around $\vec{p}$ we center a small sphere containing a number of random points. Each of these points is now checked against the depth buffer in turn. Since the G-Buffer is just a set of textures, these checks simply amount to texture lookup. If at least half of these points are visible we have full ambient illumination. Otherwise, we scale the ambient illumination proportionally to the number of unoccluded points. If we use the same set of random points for each sphere, the result will be noisy but in a very structured way. Unstructured noise is generally less objectionable, so the first improvement is to make the results more random. A simple approach is to perform a random rotation of the point set.

However, results tend to be very noisy. To alleviate the noise, most SSAO implementations use two tricks: a simple trick is to compute SSAO at half resolution. This both makes the noise more low frequency and it also improves performance. However, it is rarely enough and usually a blurring is also used. However, a simple blur filter which does not respect edges gives poor results and could cause the SSAO to "bleed" into regions where it is not intended. What is often used is a so called bilateral filter where we weight neighboring pixels with a weight that is inversely proportional to the difference between the depth value at the contributing pixel and the center pixel. Thus, we do not blur across a depth discontinuity.

8.3.1 Alchemy AO

Truth be told, however, the CryTek method is perhaps a bit dated as of writing. Fortunately, many new methods have emerged. In a recent survey by Frederik Aalund,[4] the method by McGuire et al.[5] which was created for the Alchemy game engine was found to be both most efficient and to produce a generally superior result when

[4] Frederik P. Aalund. A comparative study of screen-space ambient occlusion methods. B.Sc. Thesis, Technical University of Denmark, February 2013

[5] Morgan McGuire, Brian Osman, Michael Bukowski, and Padraic Hennessy. The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 25–32. ACM, 2011

compared to a range of other methods. In the following, we will derive and discuss a method which is very similar to Alchemy AO.

We can look at (8.1) differently by considering the occlusion instead of the amount of light. Then the integral becomes

$$\frac{1}{\pi}\int_{\Omega}(1-v(\vec{p},\boldsymbol{\omega}))\vec{\omega}\cdot\vec{n}\,d\omega = \frac{1}{\pi}\int_{\Gamma}(1-v(\vec{p},\vec{\omega}))\vec{\omega}\cdot\vec{n}\,d\omega,$$

where Γ is the part of the hemisphere Ω where $v(\vec{p},\vec{\omega}) < 1$, i.e. the only place where the integrand is non-zero. The advantage of this formulation is that we do not need to sample in the areas where the hemisphere is un-occluded but only in the areas where it is occluded.

This leads to the discrete formulation

$$\frac{1}{N}\sum_i(1-v(\vec{p},\vec{\omega}_i))\vec{\omega}_i\cdot\vec{n} = \frac{1}{N}\sum_i \max\left(0,1-\left(\frac{\|\vec{v}_i\|}{R}\right)^2\right)\max\left(0,\frac{\vec{v}_i}{\|\vec{v}_i\|}\cdot\vec{n}\right).$$

Observe that if we sample points in the G-Buffer around $\vec{p}$ we get points on the geometry of the scene. Thus, we can see the geometry part of the G-Buffer as a sampling of the domain Γ. We are making several crude approximations. Perhaps the most glaring issue is that if we sample a point $\vec{q}$ by looking at a nearby pixel in the G-Buffer, it might be that a ray from $\vec{p}$ to that point would intersect the scene before impinging $\vec{q}$. However, in practice it does not appear to be a problem.

Recalling that the equations above represent the occlusion and not the light, we finally arrive at term we can use to scale our ambient light

$$A(\vec{p}) = 1+\beta-\frac{1}{N}\sum_i \max\left(0,1-\left(\frac{\|\vec{v}_i\|}{R}\right)^2\right)\max\left(0,\frac{\vec{v}_i}{\|\vec{v}_i\|}\cdot\vec{n}\right), \tag{8.2}$$

where $\vec{p}$ is the point corresponding to our current fragment. Assuming that we are using deferred rendering, we can obtain $\vec{p}$ by doing a lookup in the geometry component of our G-Buffer at the current pixel. Performing a random sampling of N nearby pixels and finding the corresponding $\vec{q}_i$, we can compute the vector from current to nearby position $\vec{v}_i = \vec{q}_i - \vec{p}$. We also need the radius R and the (eye space) normal $\vec{n}$. Finally, $\beta \ll 1$ is a small value that we add. This constant brightens the ambient light a little bit and tends to reduce artefacts. Figure 8.4 illustrates some of these terms.

All things told, in our experience (8.2) is a very effective formulation where the structured noise does not become objectionable if we use a reasonable number of samples (e.g. 32). Thus, we advice to not decouple the sampling patterns and to leave out the final image

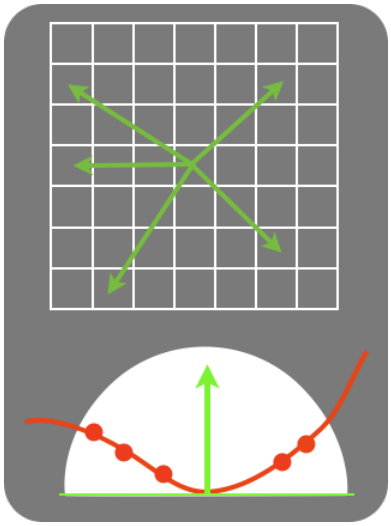


Figure 8.4: Illustration of the Alchemy AO scheme. The green arrows on top indicate points near the center pixel in the G-Buffer. Finding their positions in eye space, $\vec{q}_i$, we obtain the red points (below) shown within a white hemisphere of radius R. From these, we can compute the $\vec{v}_i = \vec{q}_i - \vec{p}$

based smoothing of the SSAO. On the other hand, we do compute the SSAO at half resolution since it is a fairly smooth signal.

8.4 Learning More

There is a vast literature on methods for shadows and ambient occlusion. Unfortunately, this problem reflects the fact that the more correct approaches are very demanding computationally. Thus, many hacks have been devised. In particular, all SSAO approaches are truly hacks. On the other hand, they are frequently used since more principled approaches for the same effects are simply not efficient enough.

As usual, Real-Time Rendering[6] is a good place to look for an overview of methods including shadow volumes which work quite differently from shadow maps and are also used a great deal. Shadow maps were originally introduced by Lance Williams in 1978 [7] and the principles are still largely the same. Perhaps the biggest issue with shadow maps is that using just one rarely gives a satisfactory result. Parallel Split Shadow Maps [8] and Cascaded Shadow Maps [9] are both methods which seek to remedy this issues by using several maps that each cover a different part of the view frustum.

The survey by Frederik Aalund[10] provides an introduction to SSAO methods and also gives a nice comparison of the methods.

[6] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 2008

[7] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978

[8] Fan Zhang, Hanqiu Sun, and Oskari Nyman. Parallel-split shadow maps on programmable gpus. *GPU Gems*, 3:203–237, 2007

[9] Rouslan Dimitrov. Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007

[10] Frederik P. Aalund. A comparative study of screen-space ambient occlusion methods. B.Sc. Thesis, Technical University of Denmark, February 2013

8.5 Project: Shadows and Screen Space Ambient Occlusion

Objectives

Your objective is to implement the screen space ambient occlusion method described in this chapter. Most of the implementation has been made. You just need to fill in some missing pieces. To this you add shadows using the traditional shadow mapping approach.

Practicalities

The code for this exercise is purely GLSL and goes into `render_deferred_ssao` and `deferred_ssao_combination.frag`. Especially the former of these two shader files have a lot of uniform and constant variables

that need to be taken into account: `disc_points` is a set of discrete 3D points that actually lie in a 2D disc, `NO_DISC_POINTS` is the number of such points, `RADIUS` is the radius R from the Alchemy method for SSAO, and `WIN_SCALING` is the scaling factor between the size of the window used for SSAO and the real on-screen frame buffer.

`WIN_RADIUS` can be used as the radius in window space inside which we sample. However, ideally, the window radius should be inversely proportional to the distance to the eye.

Part 1

Read through `render_deferred_ssao` and describe the render passes in the function.

Part 2

First add normal per pixel Phong shading in `deferred_ssao_combination.frag` (as you have done before). Next, in `deferred_ssao.frag`, the loop which tests points against the G-buffer is missing. Fill in the missing parts. Also output the ambient light intensity. Experiment with the constants `RADIUS` and `WIN_RADIUS`. Try to tweak these parameters to obtain the best quality. Discuss your selection of parameters. Take some time to look at the images produced. Try to spot artifacts and discuss these. Could the current implementation be improved, and, if so, how? Illustrate your answers with screen shots and/or drawings.

Part 3

SSAO in itself gives a somewhat odd result. Occlusion of the light from the ambient without casting shadows from the actual light source(s) looks strange. A significantly better result can be obtained if we add a shadow map. There is a shadow map but the GLSL function which samples the shadow map in `deferred_ssao_combination.frag` is not implemented. That is now your task. Explain why it is so hard to avoid a jagged appearance of the shadow. Explain the transformation (computed in the C++) code used to transform a point from the eye space positions in the G-buffer to the shadow map.

Deliverables

Screen shots of the scene, your own shader code, and the answers and discussions in Part 1 & 2.

9
Skylight and Irradiance Environment Maps

Rendered outdoor environments rarely look realistic until some sort of sky model is included (Figure 9.1). The easiest way to include a sky in a rendering is to take an image of the sky and insert it as a background texture. This approach is however fairly limited. In the long run, it would take a large database of high dynamic range sky images to make it useful as a general approach. Another solution is to physically simulate the appearance of the sky. This is unfortunately not a trivial task. The third option, the one that we are going to pick here, is to use an analytical model. There are CIE[1] standard models for the luminance of the sky, but these are not wavelength dependent. Some analytical models that include color information have been suggested in the graphics community. We will use Preetham's sky model[2] and also set up a light source that resembles direct sunlight.

In the previous chapter (Shadows and Ambient Occlusion), the ambient occlusion term would assume that everything not occluded is white. To improve on this simplification, we will now instead of white use colours based on the environment. We do this efficiently using the popular concept of irradiance environment maps (also referred to as diffuse or prefiltered environment maps),[3] where we disregard occlusion and precompute an environment map containing the irradiance due to environment lighting for a surface point with normal $\vec{n}$. In a shader, the (unoccluded) irradiance due to the environment is quickly retrieved by a simple look-up into the irradiance environment map using the surface normal $\vec{n}$.

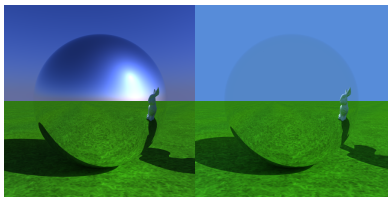


Figure 9.1: Sky model vs. no sky model.

[1] Commission Internationale de l'Eclairage, `http://www.cie.co.at/`

[2] A J Preetham, P Shirley, and B Smits. A practical analytical model for daylight. In *Proceedings of ACM SIGGRAPH 1999*, pages 91–100, 1999

[3] G S Miller and C R Hoffman. Illumination and reflection maps: Simulated objects in real environments. In *ACM SIGGRAPH 84 Course Notes for Advanced Computer Graphics Animation*, 1984

9.1 *Preetham's Sky Model*

The atmosphere is a fairly complicated layered medium consisting of many different light scattering particles such as aerosols and ozone. It can be modeled and rendered as a participating medium,[4] but this seems too expensive for real-time applications. Instead, we decide to stay inside the atmosphere and use an analytic function to obtain the skylight that arrives at a point on Earth from a given direction when the sun is in a given position. There are several analytic functions of this kind. We use Preetham's as it is relatively simple and returns photometric tristimulus values that we can convert to RGB colors.

[4] Kirk Riley, David S Ebert, Martin Kraus, Jerry Tessendorf, and Charles Hansen. Efficient rendering of atmospheric phenomena. In *Proceedings of the EGSR 2004*, pages 375–386, 2004

The model depends on the solar position (θ_s, ϕ_s) and the turbidity T of the atmosphere ($T = 2$ is clear sky, $T = 10$ is hazy sky). The solar inclination angle θ_s can be calculated by placing the scene at a latitude and specifying the Julian day (the day of the year in 1 to 365) and the time of the day (in 0 to 24). We refer to the original paper by Preetham et al. for further details. Example renderings using the sky at two different times are in Figure 9.2.

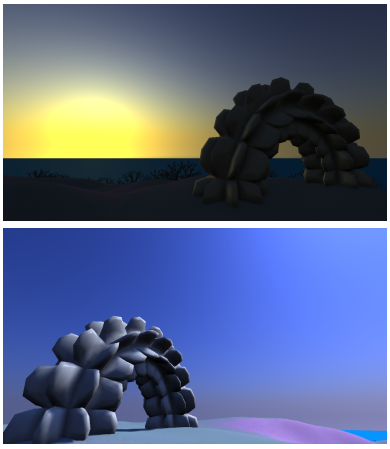


Figure 9.2: The terrain scene at two different solar times of Julian day 200 at the 45°N circle of latitude (5 o'clock at the top and 10 o'clock at the bottom).

From Preetham's sky model, we have a function that returns the sky color as a function of the view direction and a function that returns the sun color. Drawing the solar disk in the sky and sampling it would be too expensive in a real-time application. Instead, we approximate the direct sunlight using a directional light. The sun color corresponds to radiance directly transmitted through the atmosphere, which leads to some irradiance incident on the surface of the Earth due to direct sunlight. This irradiance is determined by looking at all directions above a surface point and getting a contribution only from the directions that point directly toward the sun. As the sun is far away, these directions are relatively few. In other words, the solid angle subtended by the solar disk as seen from Earth is small (around $6.74 \cdot 10^{-5}$ steradians). A directional light has an infinite area, but when using it to model direct sunlight, it should produce the same irradiance at the surface of the Earth. We achieve this by setting the radiance of the directional light (its diffuse color) equal to the irradiance due to direct sunlight (the sun color multiplied by the solid angle subtended by the solar disk).

To adapt the skylight for a real-time application, we precompute a sky cube map using the function that returns the sky color as a function of the view direction. The cube map consists of six faces that fold to a cube (Figure 9.3). The faces are numbered from 0 to 5 and ordered as follows: left, right, top, bottom, front, back. A direction vector $\vec{v}$ is used to perform a look-up into a cube map texture. The

direction associated with the texel at index i, j of face k is given by

$$\vec{v}_{ijk} = \vec{a}_k + \vec{u}_k \left(2\frac{i}{\text{res}} - 1\right) + \vec{b}_k \left(2\frac{j}{\text{res}} - 1\right), \tag{9.1}$$

where res is the texel resolution of a side of a cube map face and $\vec{a}_k$, $\vec{u}_k$, and $\vec{b}_k$ are the major axis direction, the up direction, and the right direction of face k. With this formula (9.1), we can get the view direction for each texel and thereby precompute the sky cube map.

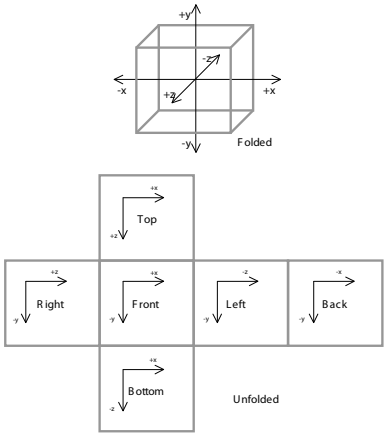


Figure 9.3: The faces of a cube map.

9.2 *Environment Mapping*

Using a cube map for environment mapping, is conceptually simple. The idea is that any eye ray (reflected or not) will see the environment stored in the cube map if it does not see an object. Conveniently, a cube map takes a direction vector as texture coordinates. Thus, to perform an environment look-up for a given eye ray, we need its direction in world coordinates. Suppose we would like to use reflection of the environment in our shader. In eye space, the eye ray direction $\vec{i}$ is simply the normalized eye space fragment position:

$$\vec{i} = \vec{p}_e / \|\vec{p}_e\|, \tag{9.2}$$

which is the same as a negation of the view vector used in Phong shading ($\vec{i} = -\vec{v}$). To get the reflection of the ray direction around the normal $\vec{n}$, we use

$$\vec{r} = \vec{i} - 2(\vec{n} \cdot \vec{i})\vec{n}. \tag{9.3}$$

GLSL has built-in functions for these operations: (9.2) `normalize(`$\vec{p}_e$`)` and (9.3) `reflect(`$\vec{i}$`,` $\vec{n}$`)`. Before using the reflection vector $\vec{r}$ as texture coordinates in a cube map look-up, we transform it from eye space to world space using the inverse view transformation matrix:

$$\vec{r}_w = \vec{V}^{-1} \begin{bmatrix} r_x \\ r_y \\ r_z \\ 0 \end{bmatrix}.$$

We can then multiply the look-up result by a specular reflectance and obtain reflective surfaces. An example is given in Figure 9.4. The limitation is that only the environment is reflected.

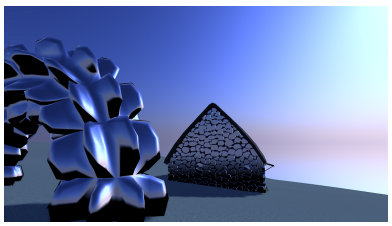


Figure 9.4: Terrain scene rendered with stone and water as purely reflective surfaces (reflecting environment only).

To draw the environment in the background, we would normally draw a screen-filling quad very close to the far plane of the view frustum. This is most easily drawn in normalized device coordinates (NDC), where the diagonal goes from $(-1, -1, 0.999, 1)$ to $(1, 1, 0.999, 1)$. When a quad is defined in NDC, the position of a

point to be shaded will also be in NDC (not in eye space). To texture map this background quad using the environment map, we thus need a transformation from a point in NDC $\vec{p}_n$ to an eye ray direction vector in world coordinates $\vec{i}_w$. Depth is not important as the eye ray direction is the same for all depths. We can therefore apply the inverse projection matrix $\vec{P}^{-1}$ directly to $\vec{p}_n$ to get the point in eye space. Having the point in eye space, we only need to convert it to a direction vector (set the w-coordinate to zero) and apply the inverse view matrix. We include the conversion to direction vector in the transformation matrix by only using the rotational part of the inverse view matrix. Then

$$\vec{i}_w = \begin{bmatrix} & & & 0 \\ & \left(\vec{V}^{-1}\right)^{3\times 3} & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \vec{P}^{-1}\,\vec{p}_n\,.$$

In deferred shading a screen-filling quad is always drawn, so we need not draw one to insert the background. Instead, we clear the z-coordinate of the G-buffer to something further away than the far plane (-1000, say). Background pixels are then pixels in the G-buffer with no geometry, that is, pixels still having the z-coordinate that they were cleared to. We only have pixel coordinates $\vec{p}_p = (x_p, y_p, z_p)$ for these pixels (`gl_FragCoord`). By applying an inverse viewport transformation to the pixel coordinates (where depth is unimportant), we get a point in NDC:

$$\vec{p}_n = \begin{bmatrix} 2\frac{x_p}{W} - 1 \\ 2\frac{y_p}{H} - 1 \\ 0 \\ 1 \end{bmatrix},$$

which we insert in the equation above to get the eye ray direction. The parameters W and H are the window dimensions. Section 2.2 has more details on the different pipeline transformation that we partially invert here.

9.3 *Sampling Skylight*

The light reflected in the direction $\vec{\omega}_o$ from a surface point $\vec{x}$ with normal $\vec{n}$ is the following integral over all light incident from the hemisphere Ω centered on $\vec{n}$:

$$L_r(\vec{x}, \vec{\omega}_o) = \int_\Omega f_r(\vec{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\vec{x}, \vec{\omega}_i)(\vec{n} \cdot \vec{\omega}_i)\, d\omega_i\,,$$

where f_r is the bidirectional reflectance distribution function (BRDF) and L_i is the light incident from the direction $\vec{\omega}_i$. This integral was

also briefly discussed in a less general form in the chapter on Shadows and Ambient Occlusion. As when working with ambient occlusion, we assume that materials are perfectly diffuse. This means that the BRDF is simply a constant:

$$f_r(\vec{x}, \vec{\omega}_i, \vec{\omega}_o) = \frac{\rho_d}{\pi},$$

where ρ_d is the diffuse reflectance of the material (the diffuse color).

Environment lighting means that light is incident from all unoccluded directions. In the case of ambient occlusion, we replaced the incident light (L_i due to the environment) by the visibility term. When considering skylight, we neglect the visibility term and replace the incident light L_i by a look-up into the sky cube map L_{sky}. The integral we need to solve is then

$$L_r(\vec{x}, \vec{\omega}_o) = \frac{\rho_d}{\pi} \int_\Omega L_{\text{sky}}(\vec{\omega}_i)(\vec{n} \cdot \vec{\omega}_i)\, d\omega_i = \frac{\rho_d}{\pi} E_{\text{sky}}(\vec{n}), \tag{9.4}$$

where we refer to E_{sky} as the sky irradiance.

We solve the irradiance integral using Monte Carlo integration. This is done by sampling directions according to a probability density function (pdf). The Nth estimator of the integral is then given by

$$E_{\text{sky},N}(\vec{n}) = \frac{1}{N} \sum_{j=1}^{N} \frac{L_{\text{sky}}(\vec{\omega}_{i,j})(\vec{n} \cdot \vec{\omega}_{i,j})}{\text{pdf}(\vec{\omega}_{i,j})}.$$

With larger N, we reduce the noise in the estimate of the integral. Conveniently, we can sample directions on a cosine-weighted hemisphere using

$$(\theta_i, \phi_i) = (\cos^{-1}\sqrt{1-\xi_1}, 2\pi\xi_2) \tag{9.5}$$

$$(x, y, z) = (\cos\phi_i \sin\theta_i, \sin\phi_i \sin\theta_i, \cos\theta_i), \tag{9.6}$$

where ξ_1, ξ_2 are canonical uniform random variables in $[0, 1)$. The resulting direction (x, y, z) is in a local coordinate system where the z-axis is the normal. To transform the sampled direction to world space where $\vec{n} = (n_x, n_y, n_z)$ is the normal, we use the following efficient formula[5]

$$\vec{\omega}_{i,j} = \left(x \begin{bmatrix} 1 - n_x^2/(1+n_z) \\ -n_x n_y/(1+n_z) \\ -n_x \end{bmatrix} + y \begin{bmatrix} -n_x n_y/(1+n_z) \\ 1 - n_y^2/(1+n_z) \\ -n_y \end{bmatrix} + z \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \right). \tag{9.7}$$

The pdf of the cosine-weighted hemisphere is

$$\text{pdf}(\vec{\omega}_{i,j}) = \frac{\vec{n} \cdot \vec{\omega}_{i,j}}{\pi}.$$

[5] Jeppe Revall Frisvad. Building an orthonormal basis from a 3d unit vector without normalization. *Journal of Graphics Tools*, 16(3):151–159, August 2012

The beauty of this sampling scheme is that the final estimator then becomes

$$E_{\text{sky},N}(\vec{n}) = \frac{\pi}{N} \sum_{j=1}^{N} L_{\text{sky}}(\vec{\omega}_{i,j}) \,. \tag{9.8}$$

9.4 *Irradiance Environment Maps*

If we store E_{sky} in a cube map, we have an irradiance environment map that can be used at runtime to get an improved ambient term.[6] The irradiance environment map is used by look-ups in the normal direction, so we precompute the irradiance environment map by evaluating $E_{\text{sky},N}(\vec{v}_{ijk})$, where $\vec{v}_{ijk}$ are the directions of the cube map texels (9.1). We can conveniently perform this computation on the GPU using layered rendering.[7] In layered rendering, the geometry shader spawns the same geometry into multiple framebuffer layers. We need six layers to render the sky irradiance map. The layer index is then the cube map face index k and the pixel coordinates (`gl_FragCoord.xy`) are the texel coordinates i, j.

[6] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986

[7] `https://www.opengl.org/wiki/Geometry_Shader`

Random numbers are required to evaluate $E_{\text{sky},N}(\vec{v}_{ijk})$ in the fragment shader that renders the sky irradiance map. With integer arithmetic, we construct a linear congruential generator (lcg) that provides pseudo-random numbers.[8] Each fragment should use a different seed for the lcg. The seed number is stored in an unsigned integer variable and passed to the floating point random function (`rnd`). This function updates the integer using the lcg and returns the next random number in $[0, 1)$. With this random number generator, we can sample directions of incidence (9.5–9.7) and use these for looking up incident skylight L_{sky} when evaluating the Nth sky irradiance estimator (9.8).

[8] Donald E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1998

To have an acceptable noise level in our sky irradiance map, we need a rather large N. However, since we compute the map on the GPU, a large N may stall the display system for a while. This is problematic as some display drivers shut down programs that stall the display system for too long. Furthermore, we may want to change the time of day interactively and would prefer not to wait for the sky irradiance map to converge at every change. To solve this problem, we use a small N and progressively update the sky irradiance map over a number of frames. If the current result in a texel is $E_{\text{sky},m}$, we

update it with a new result $E_{\text{sky,new}}$ by

$$E_{\text{sky},m+1} = \frac{E_{\text{sky,new}} + mE_{\text{sky},m}}{m+1}\,.$$

To access to the current result when computing the next, we use two buffers and read from one while writing to the other. After each pass, the two buffers switch roles so that the one we wrote to becomes the one we read from and vice versa. This is called a ping pong scheme.

The irradiance environment map is now available for use in other shaders. This is true even if the texture is still being updated between frames. As we use a directional source to model direct sunlight, we use the sky irradiance map in combination with the classic Phong illumination model (Section 2.4.1). The directional source is used in the usual way in the diffuse and specular terms of the Phong model. Diffusely reflected skylight (9.4) replaces the ambient term of the model. Finally, we can augment the model by an extra specular component, which is the skylight reflected toward the viewer and scaled by a specular reflectance coefficient. This is valid as the specular term of the Phong model only includes reflection of the direct sunlight. In an equation, the illumination model becomes

$$L_r = \frac{\rho_d}{\pi}\left((\vec{n}\cdot\vec{l})VL_{\text{sun}} + E_{\text{sky}}(\vec{n}_w)\right) + \rho_s\left((\vec{r}\cdot\vec{l})^p VL_{\text{sun}} + L_{\text{sky}}(\vec{r}_w)\right), \tag{9.9}$$

where L_{sun} is the radiance of the directional light, V is the visibility obtained from shadow mapping, $\vec{l}$ is the direction toward the sun, $\vec{n}_w$ is the normal in world coordinates, p is a shininess, and ρ_s is a specular reflectance. The remaining quantities are as defined previously in this chapter.

As a final refinement of the model, we combine it with screen space ambient occlusion. Recall that the integral (9.4) defining the sky irradiance E_{sky} neglects visibility. Conversely, the ambient occlusion scale A evaluates the same integral including visibility but neglecting variation in the environment lighting. The following crude approximation of the irradiance integral includes both visibility and environment lighting:

$$\begin{aligned} E(\vec{x},\vec{n}) &= \int_\Omega V(\vec{x},\vec{\omega}_i)L_{\text{sky}}(\vec{\omega}_i)(\vec{n}\cdot\vec{\omega}_i)\,d\omega_i \\ &\approx \frac{1}{\pi}\int_\Omega V(\vec{x},\vec{\omega}_i)(\vec{n}\cdot\vec{\omega}_i)\,d\omega_i \int_\Omega L_{\text{sky}}(\vec{\omega}_i)(\vec{n}\cdot\vec{\omega}_i)\,d\omega_i \\ &= A(\vec{x})E_{\text{sky}}(\vec{n})\,. \end{aligned}$$

This is accurate for one-color environment lighting and in cases of full visibility or full occlusion. For mid range visibility, the sharper and more localized the peaks of the environment lighting, the more inaccurate the approximation. Since direct sunlight is not a part of

our environment lighting, we find it acceptable to use E instead of E_{sky} in our illumination model (9.9).

9.5 Learning More

An interesting concept is dynamic irradiance environment maps (Gary King, Chapter 10 of *GPU Gems 2*, Real-Time Computation of Dynamic Irradiance Environment Maps, 2005), where the irradiance environment map is quickly recomputed at runtime. Finally, mipmapping (M. Ashikhmin and A. Ghosh, Simple blurry reflections with environment maps, Journal of Graphics Tools 7(4):3–8, 2002) or other sampling schemes can be used to obtain prefiltered environment maps that are useful for glossy reflections.

9.6 Project: Skylight

Objectives

Your objective is to add time-dependent skylight and sunlight to the terrain scene. Skylight will replace the flat background color and the flat ambient light. Sunlight will lead to a better directional light that follows the position of the sun in the sky. The new improved ambient term will depend on the appearance of the sky environment. Modulating this environment irradiance by screen space ambient occlusion leads (in combination with the other effects) to a nice real-time rendering method for outdoor scenes.

Part 1

In `render_deferred_ssao_sky`, we do mostly the same as in `render_deferred_ssao`. The main differences are that we now use Preetham's sky model to set the directional source and to precompute cube map textures. One of the cube maps is the sky environment of a particular day and time of the year at a specific latitude of the Earth, and it is recomputed if we change the time of the day by pressing 'h' or 'H' on the keyboard (plus/minus half an hour).

In `deferred_ssao_combination_sky.frag`, the full environment map is drawn in the background (pixels in the G-buffer with no geometry) as in Figure 9.5. Make the background sky follow the view by transforming `gl_FragCoord` to eye ray directions in world space. First, convert your pixel coordinates $\vec{p}_p$ to normalized device coordinates (NDC) $\vec{p}_n$. Then use the provided matrix `ss_proj_view_inv` to transform from $\vec{p}_n$ to an eye ray direction vector in world coordinates $\vec{i}_w$, and use $\vec{i}_w$ for look-up into the cube map.

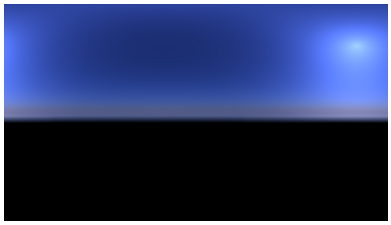


Figure 9.5: Sky environment example drawn in latitude-longitude format.

To shade the scene (pixels in the G-buffer with geometry), implement the parts of the illumination model (9.9) that involve direct sunlight L_{sun}. In the deferred shading, the rgb components of the base color correspond to ρ_d, the diffuse light is L_{sun}, and the alpha component of the base color multiplied by the specular light is ρ_s. When done, light and shadows due to the directional light should correspond to the position of the sun in the sky.

Part 2

The next step is to compute an irradiance environment map to be used as an improved ambient term. Just like the sky cube map, the irradiance environment map is also a cube map. The main difference is that we compute this cube map on the GPU using layered rendering. Set this up in the geometry shader (`integrator.geom`) by using the built-in variable `gl_Layer` to redirect the output to the different faces of the cube map. This variable has to be set before each call to `EndPrimitive()`. Pass the layer index to the fragment shader using an integer output. The triangles remain unchanged apart from the fact that they are output six times.

Use Monte Carlo integration to compute the irradiance of each cube map texel processed by the fragment shader (`integrator.frag`). The direction vector corresponding to a texel should be used as the normal in the irradiance computation. Sample a cosine-weighted hemisphere around the normal and find the incident radiance for each sampled direction by a look-up into the sky cube map. To ease the debugging process, we recommend that you render the irradiance environment map as background (instead of the sky cube map) while solving this part. The result should be similar to Figure 9.6.

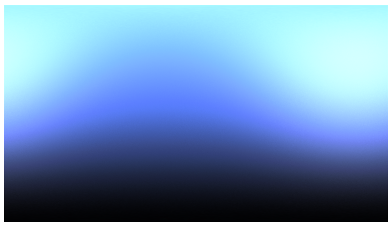


Figure 9.6: Irradiance environment map for the sky in Figure 9.5.

Part 3

Switch back to using the sky cube map as background. Now, evaluate the complete illumination model (9.9) by incorporating look-ups into the sky texture, which returns L_{sky}, and the sky irradiance texture, which returns E_{sky}. Make sure that you use direction vectors in world coordinates when looking up into these cube map textures. Finally, incorporate screen space ambient occlusion by using the irradiance $E = AE_{\text{sky}}$ instead of the sky irradiance E_{sky}. Try to illustrate with rendering examples how the different terms in the model influence the object appearance.

Deliverables

Renderings of the terrain scene under different skies. Include renderings of the irradiance environment maps. Provide rendering examples that illustrates how each term in the illumination model influences object appearance. A well-chosen view of the stone portal rendered with more and more terms added is one way to do this.

10
Mapping Details: From Bump Maps to Displacement Maps

In this chapter, we concern ourselves with real-time rendering of detailed surfaces. Clearly, any sort of realistic rendering requires high quality shaders, but coarse triangle meshes also lead to several indicators that an image is computer graphics and not reality. For a long time it has been possible to mitigate the effects of a coarse triangle mesh by adding a normal map to the surface. Normal maps (or bump maps) which we consider in this chapter are a very effective way of cheating: in the interior parts of surfaces it can be hard to tell how detailed a surface is because the lighting corresponds to a more detailed surface. However, the surface is still flat. For medium level surface structure, a good solution is to add parallax mapping. There are many variations of this technique. The one we advocate often goes by the name of steep parallax mapping. With parallax mapping, objects such as irregular brick or stone walls become much more realistic, and even silhouettes can be made to look better with some variations of parallax mapping. However, displacement mapping is arguably the cleanest and simplest solution. Here we tessellate a surface to a fine mesh and displace the vertices of that mesh. We will also consider this method.

Of course, mapping of details cannot just be applied blindly, we need to consider when to have a detailed object and when to use just a coarse object. The detailed objects will always be more costly to render, and if there are many far away detailed objects in a scene, it may prove impossible to render the scene, unless we make the far away objects less costly to render. What we have just described is known as Levels of Detail (LoD) which will be our starting point in this chapter.

10.1 *Levels of Detail*

If the scene we are trying to render is small, our task is simple. The graphics card will take care of visibility for us, and we simply render everything. If we are creating a game, we might even design the scene (or level) to be of a size that we can render efficiently without concerning ourselves with efficiency. However, what about a giant landscape? Or a jungle? Or a big city model? In many cases, the answer is to exploit visibility as discussed in the chapter on efficient rendering.

Frustum culling is our first line of defense which removes objects whose bounding volumes are not inside the pyramidal region that we can see. Considerably harder but still feasibly, we might want to use occlusion culling and remove objects that are inside the frustum but hidden behind occluders. Finally, in some cases we may have objects that are inside the frustum and visible but so detailed and so many that we cannot render them efficiently. This is where we have to resort to LoD methods. The three cases are shown in Figure 10.1.

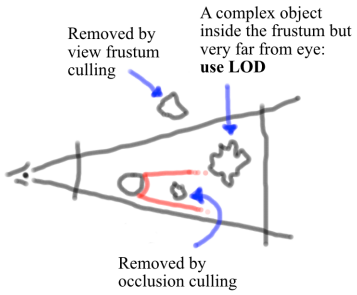


Figure 10.1: This figure shows where visibility tests can be used to reduce work and where we have to use LoD: objects that have complexity, are far away, and are insider the frustum, yet unoccluded.

LoD techniques can be classified in several ways. The most fundamental distinction is probably whether we use a *discrete* or *continuous* LoD hierarchy. In a discrete LoD hierarchy, we simply have a number of representations of each object, and we render the representation that is most suitable based on the distance from the eye position to the object. This is very straight forward, and the three bunnies in Figure 10.2 illustrate the basic principle. As we can see from the wireframe, the bunnies contain contain different numbers of triangles. The closest bunny contains around 70000 triangles and only when we are very close does it make sense with such a dense representation. The next bunny contains just a few thousand triangle, and the final one just a few hundred triangles. However, when they are sufficiently far away (as in the figure) we cannot really tell the coarse meshes from the dense.

The main problem with discrete LoD hierarchies is that the user tends to move around. Thus, the coarse bunny might look fine, but then we move closer to it and need to replace it with a more detailed version and then "pop" the user (or player) notices a slight visual pop when the coarse model is replaced with a more detailed model. Such popping artifacts are still easy to spot in otherwise very realistic computer games. The safest thing to do is to switch to the more detailed model while the visual difference (the actual difference in rendered pixels) between the fine and the coarse model is negligible. However, there is clearly a trade off between what is gained by LoD and how convincing it looks.

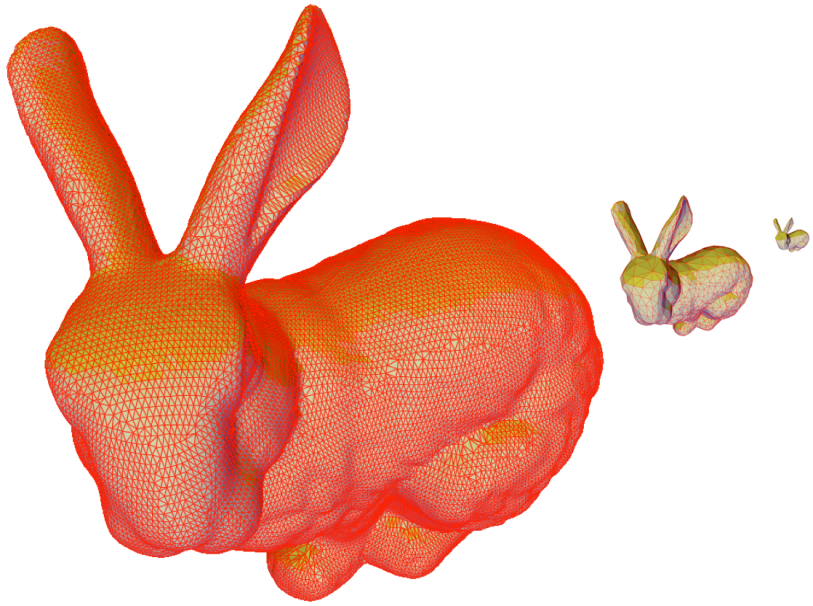


Figure 10.2: The Stanford Bunny at decreasing Level of Detail and increasing distance.

A more sophisticated approach than just replacing a coarse model with a detailed model is to use *morphing* (sometimes called *geomorphing* in the context of LoD). In general, morphing is the process where one shape is continuously changed into a different shape. For instance, we could morph an object at a give level of detail into the same object at the next coarser level. If we do the process in reverse when we go from a coarse to a fine LoD, the visual pop is replaced by a less objectionable morph.

Clearly, we can also render the two LoDs independently and then blend the images as opposed to using morphing. One interesting variation of this method is *dither fading* where a screen space pattern is used to select which of the two levels of detail that is rendered for any particular pixel. A significant advantage of this approach is that it works with deferred rendering unlike alpha blending. Also, there is no need to read the existing alpha value as we need in order to do blending.

Continuous LoD goes further than morphing. The idea is that the mesh is simply a function of distance. It might seem very challenging to create a mesh that can be parametrized in this way, but progressive meshes does just that.

There is a class of algorithms for mesh simplification which remove a vertex at a time. The best known is probably Garland and Heckbert's QSlim algorithm which operates by edge collapse. An edge collapse can be pictured as moving a vertex along one of its in-

cident edges until it hits the other vertex incident on the same edge – at which point the edge is said to have been collapsed. Clearly, this operation removes a vertex and two triangles from the mesh. Also, the triangles which contained the removed vertex need to be updated. This process can be run iteratively, removing a vertex at a time until we reach a very simple model. Running the process in reverse, we ultimately get back the original mesh.

Thus, a progressive mesh is just a mesh comprising a list of vertices and a list of triangles together with an ordered sequence of edge collapses where the ordering is according to error – the collapses which changes the geometry of the mesh least are performed first. For each edge collapse, this sequence contains the triangles that are changed when we either perform the collapse or its inverse.

Of course, the global ordering of edge collapses is not quite ideal. Ideally, we would want the part of the object that faces away from us to be as coarse as possible and the part facing toward us to be appropriately detailed. In other words, we wish to take the eye position into account. The class of techniques that does take eye position into account is called *view dependent LoD*.

Unfortunately, a problem with continuous LoD methods is that they are difficult to implement on a GPU and a CPU implementation is not fast enough.

10.2 Terrain LoD

There is one particular, frequently occurring object that does require view dependent LoD, namely terrain. By terrain, we understand an object that can be described by a height function $h(u, v)$. In other words, we cannot have caves or overhangs in a terrain, but they are frequently enormously detailed.

If we see a huge terrain as one big object, it is clear that if rendering the entire terrain is not an option, we need to choose an LoD algorithm that does take the eye position into consideration.

In the following, we will discuss a GPU friendly terrain rendering algorithm that is view dependent. Whether it is also continuous is perhaps a matter of interpretation. In any case, it is largely Thatcher Ulrich's algorithm.[1]

[1] http://tulrich.com/geekstuff/chunklod.html

The basic idea is to recursively divide the terrain into four pieces. Thereby, we obtain a tree: the top level of the tree corresponds to the entire terrain. Below the top level, we have four children – one for each quadrant – and so on. With each node of the tree, we associate a triangle mesh with the part of the terrain that the node represents.

We also store the geometric error of the terrain representation, i.e. a measure of how much it differs from our finest representation.

When rendering the terrain, we simply descend the tree recursively. For a given node, we ask whether the error is acceptable given the eye position. Clearly, for nodes farther away a greater error is acceptable than for closer nodes. If the error is acceptable, we render the terrain associated with the current node. Otherwise, we descend to its children.

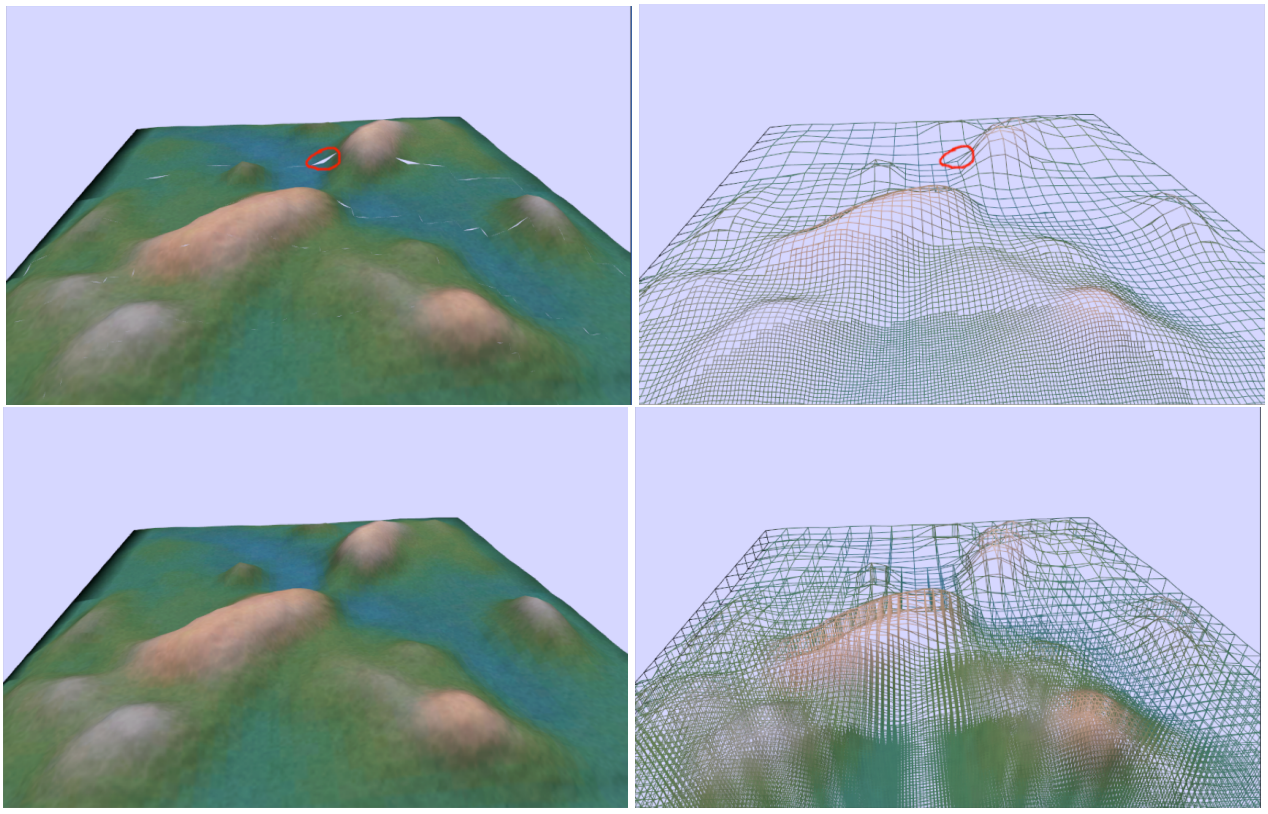


Figure 10.3: Terrain LoD. The top row shows the terrain rendered without skirts and the bottom row shows the terrain rendered wight skirts. On the right, the rendering is in wireframe.

The top row of Figure 10.3 (top) shows the method. It is clear that patches close to the eye position are finer than those farther away. In the figure, the patches are just a single quad for illustration purposes. In reality, we would use patches of hundreds if not thousands of polygons. While it is simple to have a mesh for each node, one could also use the exact same triangle mesh for all patches and simply scale it to the size of the tree node being rendered. The z coordinate for each vertex might be obtained from a texture.

Unfortunately, there are gaps between adjacent patches when they are at different levels of detail. Very complex methods can and have been devised to fix these gaps. In practice, simple *skirts* work

well. The idea is that each terrain patch is surrounded by a vertical triangle strip which has no other purpose than to close the holes in the image when rendering the terrain. The same terrain rendered with skirts is shown in Figure 10.3 (bottom). The method seems to work surprisingly well. It is impossible, in general, to detect where in the image we see terrain patches and where we see their skirts. The reason is that the vertices at the bottom edge of the skirt are rendered with same attributes (color, texture, normal) as the vertices at the top edge of the skirt. Only their positions differ.

moreover, this simple terrain rendering algorithm is fairly GPU friendly, if all terrain patches are stored in the GPU memory, and we do not use too small patches.

10.3 Normal Mapping

Textures come with their own LoD mechanism, namely mipmapping and more recently anisotropic texture mapping. As we move away from an object, we automatically start using the down-filtered, coarser levels of the mipmap pyramid.

If we were to use texture mapping to add geometric detail to an object in the fragment shader, we could simply rely on mipmapping for LoD. Note also that when the object is far away, we have fewer pixels to shade. Thus, a method which adds geometric details – somehow – in the fragment shader is going to be *output sensitive*. The fewer pixels covered, the less cost to render, which is really what we want from LoD.

Arguably, normal mapping does not add geometric detail as such, but it certainly changes the shading, making it seem that there is more geometric detail. We could simply store full 3D normals in a texture map and transform them using the normal matrix per pixel. In practice, we usually just store offset vectors that are used to change the normal or simply a height map for the surface. In the following, we will assume that the texture is a height map. Thus, while we render a piecewise flat surface, we want to convey the impression that the surface is really displaced in the normal direction. The method that we discuss in the following works well in shaders but the mathematics was derived by Jim Blinn in 1978.[2]

[2] James F. Blinn. Simulation of wrinkled surfaces. *ACM Siggraph Computer Graphics*, 12(3):286–292, 1978

Given any surface, we can compute the normal as the cross product of two tangent directions. If we assume for the moment that we have a parametric surface $\vec{S}(u, v)$, the normal is

$$\vec{N}(u, v) = \frac{\frac{\partial \vec{S}}{\partial u} \times \frac{\partial \vec{S}}{\partial v}}{\| \frac{\partial \vec{S}}{\partial u} \times \frac{\partial \vec{S}}{\partial v} \|}$$

where you can think of the surface, $\vec{S}(u,v)$, as a function that returns a point in space given a point in 2D. $\frac{\partial \vec{S}}{\partial u}$ and $\frac{\partial \vec{S}}{\partial v}$ are the partial derivatives with respect to u and v. $\frac{\partial \vec{S}}{\partial u}$ and $\frac{\partial \vec{S}}{\partial v}$ are functions of u and v just like $\vec{S}(u,v)$ but to simplify and follow convention, we omit (u,v). $\frac{\partial \vec{S}}{\partial u}$ is a tangent vector to the surface $\vec{S}(u,v)$. To be more precise, it is the tangent vector to the curve $\vec{C}(t) = \vec{S}(u+t,v)$, i.e. a curve of constant v. Likewise, $\frac{\partial \vec{S}}{\partial v}$ is tangent to a curve of constant u. The following may seem like an odd detour, since we do not have smooth surfaces but triangle meshes. However, the derivations lead to an approach, which is useful for normal mapping in the context of triangle meshes.

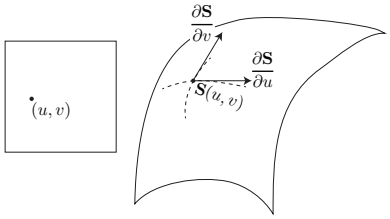


Figure 10.4: The parameter domain left and a parametric surface right.

Assume that we have a height map (aka displacement map) and use that to offset the surface:

$$\vec{S}'(u,v) = \vec{S}(u,v) + h(u,v)\vec{N}(u,v),$$

the normal of the offset surface would be

$$\vec{N}'(u,v) = \frac{\partial \vec{S}'}{\partial u} \times \frac{\partial \vec{S}'}{\partial v},$$

where we have ignored the normalization of the normal. Combining, we get

$$\vec{N}'(u,v) = \frac{\partial}{\partial u}\left(\vec{S}(u,v) + h(u,v)\vec{N}(u,v)\right) \times \frac{\partial}{\partial v}\left(\vec{S}(u,v) + h(u,v)\vec{N}(u,v)\right).$$

Things are starting to look a bit complicated, so we need to make a simplifying assumption. The first partial derivative is really

$$\frac{\partial}{\partial u}\left(\vec{S}(u,v) + h(u,v)\vec{N}(u,v)\right) = \frac{\partial \vec{S}}{\partial u} + \frac{\partial h}{\partial u}\vec{N}(u,v) + h(u,v)\frac{\partial \vec{N}}{\partial u}$$

but if we assume that $h(u,v)$ is really small, we can ignore the last term which simplifies things. Making this simplification for both the partial derivatives, we arrive at

$$\begin{aligned}\vec{N}'(u,v) &\approx \left(\frac{\partial \vec{S}}{\partial u} + \frac{\partial h}{\partial u}\vec{N}(u,v)\right) \times \left(\frac{\partial \vec{S}}{\partial v} + \frac{\partial h}{\partial v}\vec{N}(u,v)\right) \\ &= \frac{\partial \vec{S}}{\partial u} \times \frac{\partial \vec{S}}{\partial v} + \frac{\partial h}{\partial v}\left(\frac{\partial \vec{S}}{\partial u} \times \vec{N}(u,v)\right) + \frac{\partial h}{\partial u}\left(\vec{N}(u,v) \times \frac{\partial \vec{S}}{\partial v}\right),\end{aligned}$$

where one term (containing the cross product of the normal with itself) is zero and has been omitted. We recognize the first remaining term as the original normal. The second term clearly lies in the tangent plane to the surface since it is the cross product of one tangent direction and the normal. In fact, if we assume that the partial derivatives of $\vec{S}(u,v)$ are perpendicular and of the same length,

$\frac{\partial \vec{S}}{\partial u} \times \vec{N}(u,v) \approx -\frac{\partial \vec{S}}{\partial v}$. Doing likewise for the last term, and putting everything together, we obtain

$$\vec{N}'(u,v) \approx \vec{N}(u,v) - \frac{\partial h}{\partial u}\frac{\partial \vec{S}}{\partial u} - \frac{\partial h}{\partial v}\frac{\partial \vec{S}}{\partial v}. \tag{10.1}$$

This is a very practical result. $\vec{N}$ is already passed to the shader, $\frac{\partial \vec{S}}{\partial u}$, and $\frac{\partial \vec{S}}{\partial v}$ are tangent vectors which we can precompute and pass along with the normal as attributes – and also transform using the normal matrix. Together with the normal, these two partial derivatives form a coordinate frame. Often this frame of reference is denoted *tangent space* or TBN coordinates which stands for tangent, bi-tangent, and normal. We can identify with the coordinate frame used above as follows:

$$< \vec{T}, \vec{B}, \vec{N} >=< \frac{\partial \vec{S}}{\partial u}, \frac{\partial \vec{S}}{\partial v}, \vec{N} >$$

In the normal map texture, we would store $\frac{\partial h}{\partial u}$ and $\frac{\partial h}{\partial v}$. In the fragment shader, we would simply need to implement the following formula:

$$\vec{N}'(u,v) = \vec{N} - \frac{\partial h}{\partial u}\vec{T} - \frac{\partial h}{\partial v}\vec{B}. \tag{10.2}$$

where it is assumed that $\vec{T}$, $\vec{B}$, and $\vec{N}$ have been transformed according to the normal matrix.

Of course, we do not have to precompute the partial derivatives of h. But whether we do precompute or compute them in the fragment shader, this is done using central differences:

$$\frac{\partial h}{\partial u} = \frac{h(u+\Delta, v) - h(u-\Delta, v)}{2\Delta} \ , \tag{10.3}$$

where Δ is should be the distance in texture coordinates between two texels. The partial derivative in v is, of course, computed analogously. The only "gotcha" which we need to bear in mind is that h might be in the range $[0, h^{\max}]$ whereas our actual height map texture might be in the range from $[0, 1]$. In this case, it is important not to confuse the texture and the function.

Figure 10.5 shows the scene with and without normal mapping. While the difference is not huge, the normal map makes the geometry stand out much more.

Figure 10.5: The wall with just the texture map (top) and with texture plus normal map (bottom).

10.4 *Parallax Mapping*

Normal mapping works well, but it does not actually move the surface. In the formulation which we have just used, we displaced the normals using a height map. What if we ray traced that height map? In its essence that is what parallax mapping is about.

The word parallax is derived from a greek word that means alteration. In computer graphics parlance, we use it to mean how objects in a scene move as we change eye position. For a non-planar surface, a shift in eye position causes both a distortion and changes in occlusion – something may cover something else or something previously not visible may be uncovered.

Both the distortions and occlusion changes are captured by parallax mapping – at least to some extent. Let us assume that the actual triangles we are drawing represent a surface that corresponds to the maximum height of the height map. Thus, the entire surface lies below the rendered geometry. The basic setup is shown in Figure 10.6.

Now, recall that the eye space position of a fragment is simply a vector from the origin to the point in space represented by the fragment. Flipping the orientation, we get the view vector shown in Figure 10.6, but for the purpose of tracing we want to simply normalize the position, $\vec{p}$, of the fragment. Let us call the normalized position of the fragment $\vec{d} = \frac{\vec{p}}{\|\vec{p}\|}$. Note that $\vec{d}$ is still in eye space. We need to project $\vec{d}$ onto the normal, and the two tangent directions to get it in tangent space.

The transformation can be expressed as a matrix vector product:

$$\vec{d}_{\text{ts}} = \begin{bmatrix} \frac{\partial \vec{S}}{\partial u}^T \\ \frac{\partial \vec{S}}{\partial v}^T \\ \vec{N}^T \end{bmatrix} \vec{d}.$$

Say (u, v) are the texture coordinates of our fragment. Let us define the vector $\vec{u} = [u\, v\, h^{\max}]$. Classical parallax mapping now amounts to

$$\vec{u}^1 = \vec{u} + \frac{h^{\max} - h(u,v)}{\vec{d} \cdot \vec{N}} \vec{d}_{\text{ts}} \ . \tag{10.4}$$

Thus, we divide $\vec{d}_{\text{ts}}$ by its z coordinate and multiply by the vertical distance from the initial position of the ray to the height map. In the Figure 10.6 example this position is the second black dot in the middle image. In our experience, this works poorly. In fact, we will even refrain from taking a screen shot.

A much better, but sadly also more costly, solution is to use steep parallax mapping as described by McGuire and Mcguire.[3]. The idea is to step along $\vec{d}_{\text{ts}}$ until the surface is encountered. Starting from $\vec{u}^0 = \vec{u}$, we compute

$$\vec{u}^{i+1} = \vec{u}^i + s\vec{d}_{\text{ts}}, \tag{10.5}$$

where s is a scaling such that we do not skip over texels. Letting s be $\frac{1}{4}$ the texel distance should ensure that we do not trace too fast.

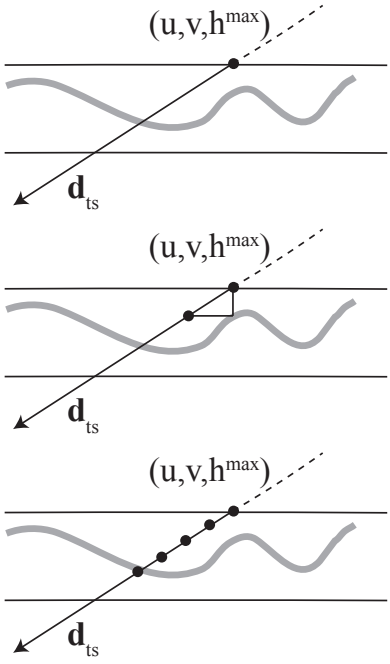


Figure 10.6: With no parallax mapping, we simply get the texture information at the point of the rendered planar surface (top). With parallax mapping, we use the height value to shift the point of texture lookup (middle). With steep parallax mapping, we simply trace until we hit the surface (bottom).

[3] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, pages 23–24, 2005

We should apply (10.5) iteratively until we either reach a maximum number of iterations or the z coordinate of $\vec{u}^i$ is less than $h(u^i, v^i)$.

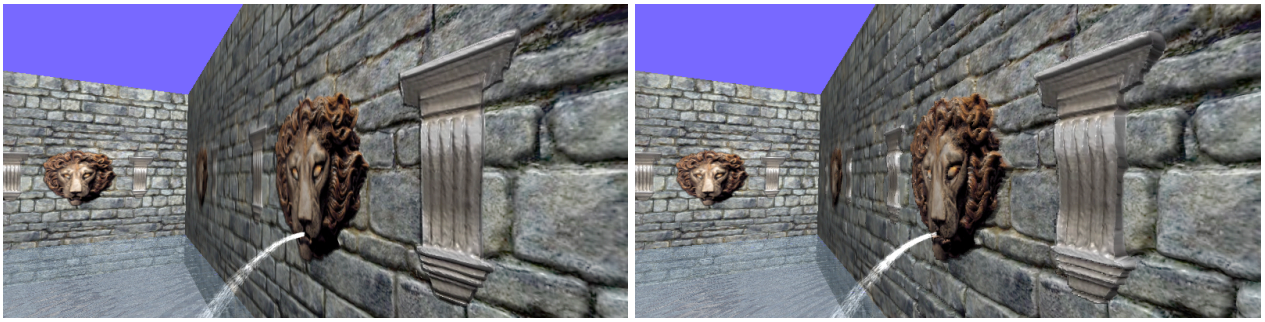


Figure 10.7: This image shows the scene with just normal mapping on the left and normal plus parallax mapping on the right.

Figure 10.7 shows a comparison of just normal mapping on the left to a combination of normal and steep parallax mapping on the right.

10.5 *Project: Steep Parallax Mapping*

Objective

In real-time rendering we often want to make flat polygons appear to have more details than they actually do. Parallax mapping (and its many variations) is basically a way of making a flat polygon appear to have details which are modeled using a height field. In this exercise, you are given a simple room scene where the walls and the floor are texture mapped with two textures: a color texture and a height map. Your goal in this exercise is to implement normal mapping and (steep) parallax mapping using the height map. Finally you shall perform some tests and evaluations. The code for this exercise is all in the `room.frag` shader.

Part 1

For each pixel, compute the normal from the depth map using central differences. Make sure that you get the normal in the correct tangent space coordinates. Use the new normal for the Phong shading used in the scene.

Part 2

Implement steep parallax mapping. Experiment with the method to get the best results. You can adjust the maximum number of ray steps, the depth scaling of the height field (given as a constant: `MAX_HEIGHT`) and the step length along the ray. Change the shader to

output a grey level image showing the number of ray steps instead of the normal color in order to test and tune the method.

Part 3

Analyze the method. Select two viewpoints - one where you are close to the wall looking along the wall and one where you are in the middle of the room. For each viewpoint render four images as described below:

- Ray step image as computed in Part 2.
- Shaded textured image. This image should be rendered with texture and shading where the normal is the original (unchanged) normal.
- Normal mapped image. Like above but using the normal from Part 1.
- Parallax mapped image. Like above but where the texture coordinates for both normal computation and texture lookup are computed using the parallax mapping from Part 2.

Time the performance for each combination of viewpoint and image type. Based on these images, analyze the importance and utility of normal mapping and parallax mapping. In particular, consider these questions:

- When (or where) do we see an improvement in image quality due to parallax mapping?
- Or an improvement due to normal mapping?
- Does the overall improvement to quality due to parallax mapping outweigh the drop in performance?
- Which is more important: Normal mapping or parallax mapping?

Deliverables

Screen shots of the scene, your own shader code, the 8 images with timings as well as the answers and discussion in Part 3.

11
Simulating Water

In this chapter, we will discuss how to simulate and render water. When a professional game engine programmer read this chapter, his only comment was: "scroll two normal maps on top of each other. Get artists to add particles until it stops looking like crap". There is some truth to that, but just like in the case of illumination there is a trend towards simulating rather than using artist man hours to make things look real. This does not necessarily work better, but - ultimately - simulation will look more real for the same amount of artist effort. Also, some effects like water ripples that reflect from the edges of a pond are easier to simulate than fake.

Water and, more generally, fluid simulation is a vast field, but we will only concern ourselves with a very particular and simple type of water simulation. The method we will discuss is based on the wave equation and has been called the "Hello World" equivalent of water simulation. It gets its simplicity in part from the fact that the water is represented as a dynamic height map.

In this chapter, we will also discuss how to render this dynamic height map so that it looks like water. In particular, this requires us to deal with both reflection and refraction.

With these two things in place, we have a one stop solution for water. At least it is possible to render a passable pond; an ocean would require more methods and different methods. However, in a later chapter, we will consider how to create a fountain and other effects using particle simulation.

11.1 Simulating Water Using the Wave Equation

In real-time graphics, we often see pools, ponds, or other small to medium sized bodies of water. For many purposes, this water does not have to be simulated with a full blown water simulation. It is very simple to create an animated ripple pattern on the water, but if the pattern does not respond to user interaction, the illusion is not

very compelling. The method that we present in the following is very simple and has been used in a number of games. It works well for small bodies of water such as a little pond, and it allows us to model waves in such bodies of water, but it does not deal with actual flow of liquid – just waves. The method is described in a bit more detail by Bridson and Müller-Fischer.[1] and it is based on a simple partial differential equation known as the *wave equation*:

$$h_{tt} = c^2 \Delta h \tag{11.1}$$

where h is the value of a height map (see Figure 11.1), and c is the speed of the propagating wave. h_{tt} on the left hand side represents the second order derivative with time. $\Delta h = h_{xx} + h_{yy}$ on the right hand side is the sum of the spatial second order derivatives of the height value. The operator Δ, which we have also used for edge detection, is known as the Laplacian.

[1] Robert Bridson and Matthias Müller-Fischer. *Fluid simulation: SIGGRAPH 2007 course notes*. ACM, 2007

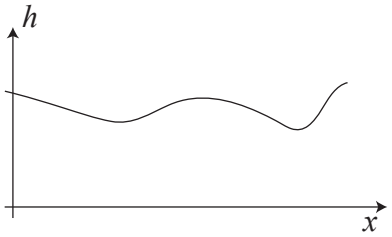


Figure 11.1: The x and h axes of a water height map. Note that the Laplacian in 1D is simply the second order derivative, which is positive at minima and negative at maxima.

While this equation looks a bit mysterious, it is fairly easy to interpret. Since $h(x, y)$ is the height of the water column at (x, y), h_t is the rate at which the height of the water column changes, and h_{tt} can be seen as the acceleration of the water – acceleration up or down that is; using this equation, we do not real model the water flow. We just model how it goes up and down.

The Laplacian is on the right hand side. Observe that in 1D, the Laplacian is simply the second order derivative and the second order derivative is positive at a minimum and negative at a maximum. That is key to understanding the wave equation: at maxima the water column has a downward acceleration and at minima it has an upward acceleration.

To better solve the wave equation, we can expand it into two equations

$$\begin{aligned} \sigma_t &= c^2 \Delta h \\ h_t &= \sigma \,, \end{aligned}$$

where we have introduced, σ, the velocity of the water column. The advantage of this rewriting is that it can be solved with what is known as a *semi-implicit Euler* solver. Plugging in the correct discretizations, we obtain the following numerical method

$$\begin{aligned} \sigma(t + \Delta t) &= \sigma(t) + \Delta t \tfrac{c^2}{\Delta s^2} \left(h(x + \Delta s, y) + h(x - \Delta s, y)\right. \\ &+ \left. h(x, y + \Delta s) + h(x, y - \Delta s) - 4h(x, y)\right) \end{aligned} \tag{11.2}$$

$$h(t + \Delta t) = dh(t) + \Delta t \sigma(t + \Delta t) \;, \tag{11.3}$$

where d is a damping constant (that makes waves die out), Δt is the time step, and Δs is the spatial grid size. The constant d should be

close to 1 but also smaller than 1, and stability requires that $c\Delta t < \Delta s$. This condition is known as the CFL condition, and if we recall that c is the speed of the wave, it really says that the time step must be small enough that the numerical propagation of the wave does not outrun the wave's actual speed. The solver is semi implicit because we first update σ and use the updated σ to compute the new h. It is worth making a note about notation: both h and σ are functions of space as well as time. However, to simplify things, we only write the parameters that are relevant in the equations above.

Of course, if we initialize h with $h_0(x, y) = 0$, the water will be perfectly calm. To create waves, we need to change the h function by manually adding a value to $h(x, y)$ for some position x, y. Clearly, we choose some x, y where we would like to make a splash.

Another thing that has not been considered is boundary conditions. When a wave hits the boundary it typically looks most natural that it is reflected. This can be attained by simply setting $h(x, y) = h(0, y)$ for $x < 0$ (and doing likewise for the three other boundaries). This condition, in turn, corresponds to using `GL_CLAMP_TO_EDGE` when specifying the texture for σ and h.

Now, we are almost ready to simulate water. From a practical point of view, we need the water to be stored in a texture. The natural way to do the update is to have a shader read from one texture and output to a different texture using a ping pong scheme.[2] Both textures can be stored in a single FBO. Each texture requires two channels – one for h and one for σ. In the implementation that accompanies this book, we have two phases to the simulation. In the first phase, we modify h (to make a splash) and then in the second phase the fluid simulation takes place. A slightly more efficient implementation would not use a render pass to add splashes to the water height map.

[2] In a ping pong scheme we use two buffers: we read from one and write to the other. After each pass, we then switch the roles so that the one we wrote to is now the one we read from and vice versa.

11.2 *Rendering a Water Height Map*

Rendering water using Phong shading would lead to unattractive results. It is very fundamental to our understanding of water that it both reflects and refracts light. The pipeline which we will use for water rendering is shown in Figure 11.2.

The first step of water rendering is to draw the scene to the back buffer quite normally. We will need that image to compute the refraction, so it is copied to a texture.

Next, the reflected scene is drawn. Since we model the water as a planar reflector this is very straight forward: a planar reflection

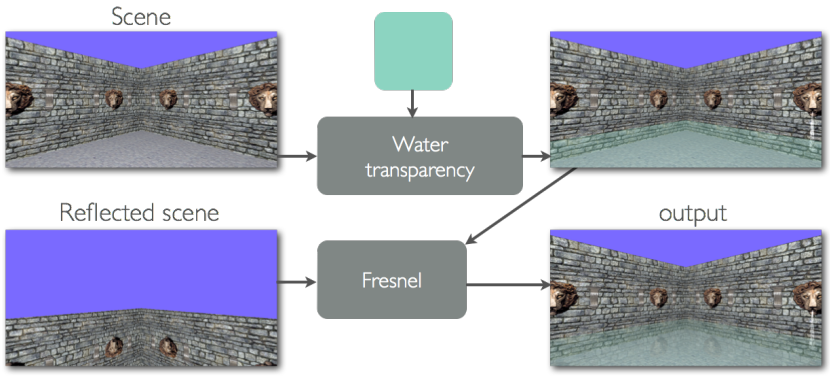


Figure 11.2: The water rendering pipeline. The initial image of the scene (top left) is combined with a constant color giving the water a plausible color (top right) to the scene as seen through water. We then combine this tinted image with the reflection, using the Fresnel term, producing the final image shown on the bottom right. Note that the intermediate stage (top right) is only for illustration purposes. Based on the original image of the scene and the image of the reflection (the two images on the left), the final result (bottom right) is computed directly in a fragment shader.

can be introduced as a part of the view matrix like any other scene transformation. If the planar reflector happens to be parallel to one of the canonical axes, it is particularly easy. The matrix that is needed to mirror in Z is

$$\vec{M} = \vec{T} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} (-\vec{T}) \ ,$$

where the translation $-\vec{T}$ is used to translate the scene so that the origin is incident on the planar reflector. $\vec{T}$ then translates the origin back. The transformation needs to be applied before the ordinary view transform, so we right multiply $\vec{M}$ onto the view matrix.

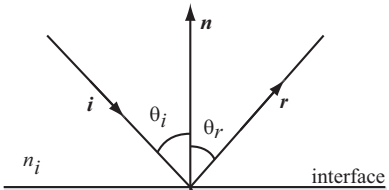
Once the view transform is set up, we render the reflection of the scene to a texture using the (by now) well known mechanism of a frame buffer object. We now have two textures that we need to combine. When light travels from one material to another, the ratio of light that is reflected is given by the formulas known as the Fresnel equations. It is easy to fake the Fresnel effect using a simple function of the angle between the surface normal and the incident ray. For instance, you can use the following as the ratio of reflected light:

$$R_{\text{approx}} = (1 - \vec{n} \cdot \vec{v})^{\alpha}$$

where $\vec{n}$ is the surface normal, $\vec{v}$ is the direction towards the eye position, and α is an exponent, say, 4. More precise approximations have been proposed, but it is not much harder to compute the actual Fresnel effect.

First, we need to compute the angle of the refracted (or transmitted) ray which is given by the law of refraction:[3]

[3] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of Photonics*. John Wiley & Sons, Hoboken, New Jersey, second edition, 2007



$$\sin\theta_t = \frac{n_i}{n_t}\sin\theta_i \ ,$$

where θ_i and θ_t are the angles of the incident and the refracted ray, respectively, n_i and n_t are the refractive indices of the media. These are constants which can be obtained from tables of materials. For the relevant materials, we have

$$n_{\text{air}} = 1.00 \ , \ \ n_{\text{water}} = 1.33 \ .$$

The refractive index of a material is the ratio of the speed of light in a vacuum to the speed of light in that material. Thus, for an interface between two materials in which light travels at very different speeds, the rays bend a lot. All of these terms are illustrated in Figure 11.3, and, in fact, we do not even need to use the above equation in many cases, since the shading language may provide a convenient `refract` function as also noted below.

Light can be polarized in two ways: perpendicular to the plane of incidence, which contains the incident, reflected, and refracted rays, or parallel to this plane. The first type is known as the *s*-polarized component (*s* for *senkrecht*, which German for perpendicular) and the second as the *p*-polarized component (*p* for *parallel*). Now, the ratios of light reflected according to these polarizations are[4]

$$R_s = \left|\frac{n_i\cos\theta_i - n_t\cos\theta_t}{n_i\cos\theta_i + n_t\cos\theta_t}\right|^2 \tag{11.4}$$

$$R_p = \left|\frac{n_i\cos\theta_t - n_t\cos\theta_i}{n_i\cos\theta_t + n_t\cos\theta_i}\right|^2 \ . \tag{11.5}$$

[4] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of Photonics*. John Wiley & Sons, Hoboken, New Jersey, second edition, 2007

This formula is easy to implement in a shader since the cosines can easily be computed using the normal $\boldsymbol{n}$ and the direction of the incident ray $\boldsymbol{i}$ (see Figure 11.3). From the law of refraction, we have

$$\cos\theta_i = -\boldsymbol{i}\cdot\boldsymbol{n} \tag{11.6}$$

$$\cos\theta_t = \sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-\cos^2\theta_i)} \tag{11.7}$$

since $\theta_i, \theta_t \in [0, \pi/2]$. Note however that if the expression under the square root ($\cos^2\theta_t$) becomes negative the result is total internal reflection ($R_s = R_p = 1$), and one should avoid taking the square root. Another option is to use the special function `refract` to find the direction of the transmitted ray $\boldsymbol{t}$ – then we simply have $\cos\theta_t = -\boldsymbol{n}\cdot\boldsymbol{t}$. For unpolarized light, we simply take the average leading to this formula

$$R = \frac{R_s + R_p}{2} \tag{11.8}$$

Thus, R is used to weight the image of the reflection, and $1 - R$ is used for the image of the background, i.e. the transmitted light. However, the result so far is still far from plausible. To begin with, we need to blend the image of the scene with a color: water is rarely completely colorless.

Water is also rarely completely still, and earlier we have taken great pains to compute a dynamic height map. Now, is the time to use that height map. The first thing we do is to compute the perturbed normal. This is done precisely as in the previous chapter where we discussed normal mapping. Clearly, we can plug the perturbed normal into the Fresnel equations. Doing so makes it possible to see that the waves are there, but it is not convincing. The air/water interface bends the light rays. When the water surface is perturbed, both the reflected and the refracted images are slightly warped. Unfortunately, it would be very computationally demanding to compute the actual reflection and refraction. In fact, it would require ray tracing, and we do not even store the depth maps for the two images.

If we allow ourselves what might be seen as an enormous hack it becomes very easy, however. While any human being can see that the reflected and refracted images should be slightly warped, nobody is able to discern whether they are warped in precisely the right way. What we propose is that you should take the partial derivatives of the height map h, i.e. $\frac{\partial h}{\partial u}$ and $\frac{\partial h}{\partial v}$ and use these to warp the texture image. Note that since they are already computed in order to perturb the normal, it is essentially costless in terms of computation. The 2D vector $[\frac{\partial h}{\partial u}\ \frac{\partial h}{\partial v}]$ is simply scaled and added to the texture coordinates. The scaling factor is very important, and it should be small, since the warp will otherwise lead to strange results. Remember that the method does not compute the actual refraction, but it looks right and nobody can tell.

Since the rendering consists of quite a few steps, we briefly summarize the procedure below in a cook book like format.

1. Render scene as normally (use as background and copy to refraction texture).
2. Render reflected scene directly to texture.
3. Render water surface. For each pixel,
   (a) compute $[\frac{\partial h}{\partial u}\ \frac{\partial h}{\partial v}]$
   (b) compute displaced normal, $\vec{N}'$
   (c) compute Fresnel term R using $\vec{N}'$.
   (d) add displacement to fragment position and perform lookup in refraction texture. Blend the result with a faint water color.

(e) add displacement to fragment position and perform lookup in reflection texture.

(f) Compute weighted average of the reflection and refraction images. Use R as the weight for the reflection and $1 - R$ for the refraction.

11.3 *Project: Water Simulation and Rendering*

Objective

Fluid dynamics is used in many places in computer graphics, and in this exercise, you will implement one of the simplest schemes which is useful for simulating the surface of a fairly shallow pool of water. In addition you will implement a simple but effective method for water rendering.

Part 1

Initially, when the water is rendered it looks like a hazy mirror. Your first task is to implement the Fresnel term which is used to blend between the reflective and the transmissive part. As shown in `render_water.frag` the Fresnel term is used to blend between the reflection stored in `ref_tex` and the transmission which is the background (from `bg_tex`) combined with the color of the water. Once the Fresnel term has been included, the water should be more reflective at grazing angles and less reflective at angles almost perpendicular to the surface. Explain what the Fresnel term is and how it is used in computer graphics.

Part 2

Make the water dynamic by implementing a numerical solver for the wave equation in the fragment program, `fluid_solver.frag`, for water simulation. Describe both the wave equation and the numerical method used for solving it. In the fragment program for water rendering, compute partial derivatives for the water height (`hf_tex`). Use these derivatives to warp the lookup position in the reflection texture (simply add the 2D vector of partial derivatives to the texture coordinates) as well as the background texture (`bg_tex`). This is a hack but it gives you a warp of the textures that is correlated with the geometry of the ripples which is the important thing. You probably want to scale the partial derivatives to reduce the amount of warping. What would be a more correct way of doing the reflection? Also compute the correct normal (exactly like in parallax mapping) and

use that for the Fresnel term. Once this is done, you should see a dynamic water surface reflecting the scene; little splashes are introduced where the jet from the fountain hits the water surface. Experiment with the constants. How is this water simulation unrealistic? Enumerate the ways!

Deliverables

Source code for all parts, answers to all questions (underlined), and screen shots illustrating the two parts.

12
Animation

Without animation, real-time computer graphics would be very dull. In many, early games (e.g. Doom) the dynamic objects were sprites (i.e. possibly animated 2D images) even though the scene might be shown in 3D. Later, static 3D meshes supplanted the sprites and, finally, vertex shaders brought dynamic meshes to real-time computer graphics. What we consider in this chapter is precisely how to make 3D triangle meshes dynamic.

We will consider two approaches: the first one is simply to have several positions for each vertex and interpolate between these. While this is easy and effective, it is a bit less general than creating a skeleton or virtual armature which we can deform as a proxy for deforming the real object. This second method, which is often called skeletal subspace deformation, is the other method which we consider in this chapter.

Our final concern is dealing with the animation of multiple objects - so called flocks of objects. The members of such flocks are often called "boids": the word appears to be constructed by contracting the longer word "bird-oid". Moreover, simply pronouncing bird with a strong accent might lead to something that sounds like "boid". Be that as it may, we will not consider the word again but discuss how to make flocks of boids behave such that they appear to be cognizant of one another.

12.1 Mesh Interpolation

The principle behind animation by mesh interpolation is not hard to understand. If we are given two meshes that have exactly the same connectivity, we can construe this is as the same mesh with different vertex positions and then we can simply interpolate the vertex positions. A slight snag is that we may have two meshes that have exactly the same vertices but if the vertices are not also numbered in the same way, it does not work. This problem was addressed by

Eppstein et al.[1]

Below, in Figure 12.1, a number of *poses* are shown for a walking male character. The poses are hand crafted from the same starting point in a 3D modeling tool, and it is easy to picture that by interpolating between these poses, we could make the character appear to walk. The principle is that we divide the animation sequence into

[1] David Eppstein, Michael T Goodrich, Ethan Kim, and Rasmus Tamstorf. Motorcycle graphs: canonical quad mesh partitioning. In *Computer Graphics Forum*, volume 27, pages 1477–1486. Wiley Online Library, 2008

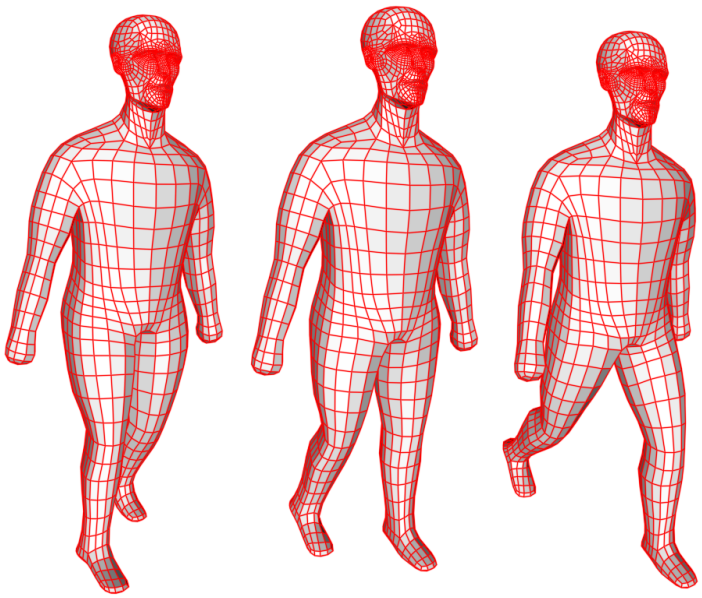


Figure 12.1: Male character in three different poses. Note that mesh has exactly the same structure in all three poses. Only the vertex positions differ between each pose.

a number of time slices. In each time slice, only two poses are active and we interpolate between those. Interpolation happens by sending a uniform time variable to the vertex shader. The vertex shader then performs a linear interpolation between the vertex position from each of the two poses:

$$\vec{p} = \frac{t_{i+1} - t}{t_{i+1} - t_i}\vec{p}_i + \frac{t - t_i}{t_{i+1} - t_i}\vec{p}_{i+1},$$

where $\vec{p}_i$ and $\vec{p}_{i+1}$ are the poses at time t_i and t_{i+1}, respectively, and t is the current time (stored in a uniform). Simple as this method is, we need to bear a few things in mind. Most importantly, we cannot just interpolate vertex positions; we need to also interpolate normals to ensure that the lighting is correct.

So far, nothing has been said about what poses we should interpolate or how to obtain poses. Figure 12.2 shows an example with nine hand drawn poses from a walk animation. These drawn poses were used to design the mesh poses shown in Figure 12.1.

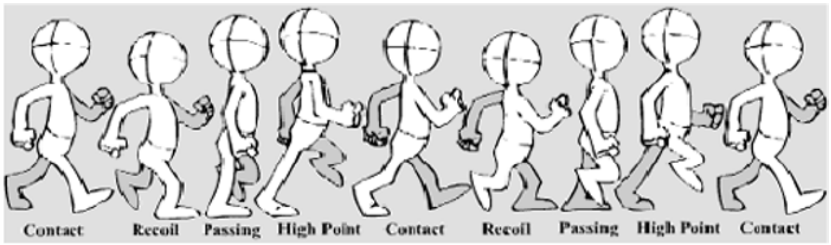


Figure 12.2: These eight poses have been chosen such that if we linearly interpolate between them, we get a plausible walk animation.

It is useful to take a moment to look at the poses in Figure 12.2. There are nine of them and only eight are different since the first and the last are identical. The first four and the last four are each other's mirror images since in the first four, the right leg touches the ground and in the next four, the left leg touches the ground. In fact, we can further reduce the number of truly unique poses to three, since the "passing" pose can be seen as a halfway between recoil and high point. This clearly begs the question: why bother having the passing pose at all. The answer is that the rhythm of the walk would be odd if the large swinging movement of the leg needed to reach the high point from the recoil pose did not have an intermediate step. This in itself could be fixed by having different time intervals between each pose. However, we also need to bear in mind that we are interpolating vertex positions linearly. The trajectory of a single vertex from one pose to the next is a straight line. There is no rotational motion: to get something that looks like a plausible rotational movement, we need to break large motions into smaller pieces. Thus, the passing pose is indispensable.

In the particular case of a walk animation, we also need to avoid that the character "moonwalks", i.e. moves his feet without his centre of mass moving correspondingly. One way of assuring this is to find a vertex on the foot that touches the ground in two poses between which the character is propelled forward. Since such a vertex is supposed to remain firmly on the ground while the foot to which it belongs pushes the character forward, we want to translate the character forward by the precise amount that the vertex moves back between the two poses. This is simple but a very important part of a convincing walk animation.

12.1.1 Procedural Animation

There is no underlying mathematical or anatomical model for the type of animation we have just discussed. However, it is possible to treat this type of animation procedurally. For a lot of old but still

very good examples, we recommend that you visit Ken Perlin's website.[2]

[2] `http://mrl.nyu.edu/~perlin/` – note that while still relevant, these examples are old, and modern, secure web browsers will refuse to load these examples unless forced. Also, some appear not to work anymore.

Like creative content generation, this requires the programmer to exercise creativity, since the animation is then defined in a program; the vertex shader to be precise. To give a simple example, say we want a character to turn his head. Looking at the model you can discover how it is positioned and oriented. For instance the z axis might point up and the origin be placed in the middle of the body. All it might require to make the head turn is a simple rotation around the z axis. Of course, only the head should rotate, so in the vertex shader, you set a threshold on the z coordinate and rotate each vertex only if the z coordinate is greater than this threshold. The one subtlety is that the head movement will probably look odd if the full rotation is applied to all vertices above the threshold and no rotation is applied to all vertices below. To solve this issue, we simply multiply our rotation angle with a smooth step function of the z coordinate to create a gradual transition.

To give another, even simpler, example, imagine that an object acts according to Newton's laws, in that case you simply find its position by integrating acceleration to obtain velocity and velocity to obtain position. If the object can simply be regarded as a particle, this is very simple. Linear elasticity - Hooke's law - can also come in handy. In fact, even in your smartphone's UI these things are sometimes used. Of course, if the use of physics becomes more complex, we would start calling it simulation. However, as long as it is simple and combined with other effects, a convincing case could be made for thinking of it as a procedural tool.

There are also uses for noise functions, clearly, in procedural animation. To give just a single example, an airborne vehicle might be rocked by turbulence using ... the turbulence function discussed in a previous chapter. Later, in the project accompanying this chapter, you will also be asked to resort to noise when it comes to animating a school of fish.

Of course, when procedurally animating the vertex positions, we also need to do the same for the vertex normals. In principle, this is not harder, but for small transformations which do not affect lighting much, it may be beneficial to ignore the change to the normal in the interest of efficiency.

12.2 *Skeletal Subspace Deformation*

While programmers sometimes have more artistic talent than they are credited with, the vertex shader is not the place to create animations. The prevailing method for animating characters in real-time graphics is to associate a virtual armature, a *skeleton*, with the 3D model. A very simple example is shown in Figure 12.3.

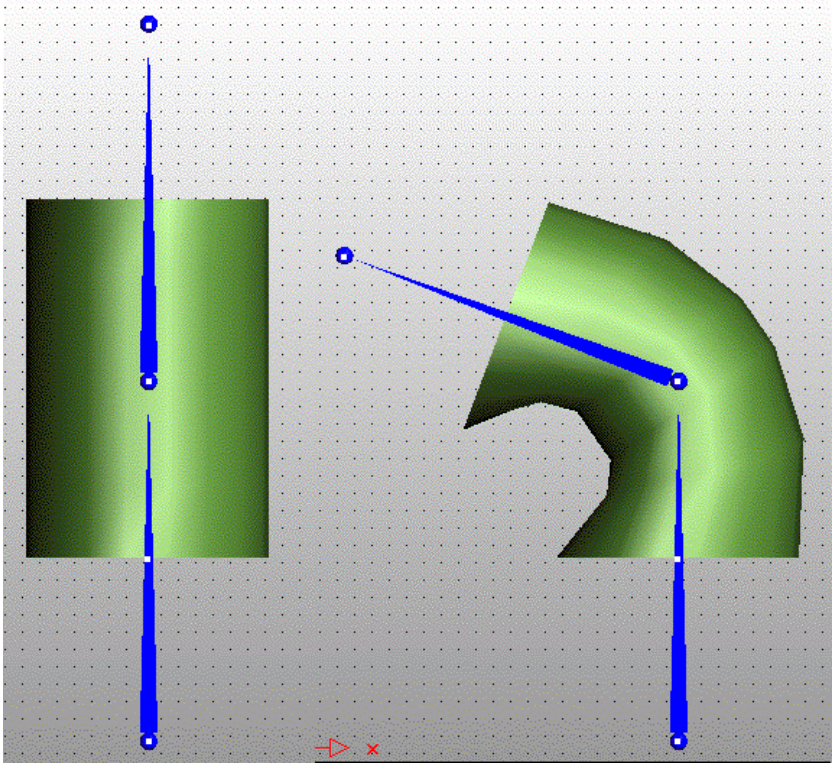


Figure 12.3: Cylinder and its associated skeleton consisting of two bones.

Each *bone* in the skeleton is usually drawn as a ball with a cone or arrow attached. In fact, what each bone represents is usually just a rotation, and the pivot is the center of the ball. Clearly, rotating a bone does not automatically rotate the skin, i.e. the mesh associated with the skeleton. The artist responsible for associating the 3D model with the skeleton needs to do something we usually call *rigging*. The basic idea is that we associate each vertex with one or more bones. In the figure, the topmost row of vertices appear to be fully associated with the topmost bone, and, likewise, the bottom row of vertices are fully associated with the bottom bone. The vertices in between are associated with both bones, but their primary association shifts gradually from on to the other. To be very precise, if a vertex is

associated 40% with bone a and 60% with bone b, the transformation matrix, we use for that vertex is $\vec{M} = 0.4\vec{M}_a + 0.6\vec{M}_b$.

The previous figure contained only two bones, in reality, we have a situation like the one shown in Figure 12.4 where the bones are organized in a hierarchy. Using the orientedness of the bones, it seems that the root of the skeleton is around the pelvis area.

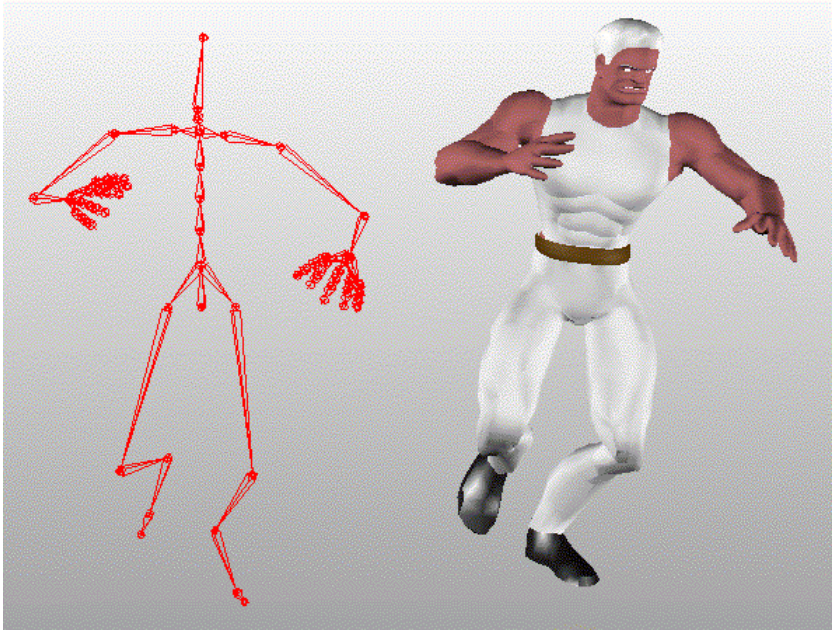


Figure 12.4: A bone structure (skeleton) and a human character in the pose induced by that structure.

Intuitively, the figure is very clear, but from a practical perspective, it may not be obvious how we make the skeleton actually move the character. First of all, the transformation of, say, the lower arm bone clearly cannot be considered in isolation; we need to compound the effect of all the bones leading from the pelvis to the lower arm.

The principle is the same as what we might use to create the robot arm shown in Figure 12.5. The code used to draw the robot arm is shown in the listing below:

```
LightPosition 0 0 1 0
ViewTransform
BeginList Joint
{
  Sphere
  Translate 0 2.5 0
  Scale 1 5 1
```

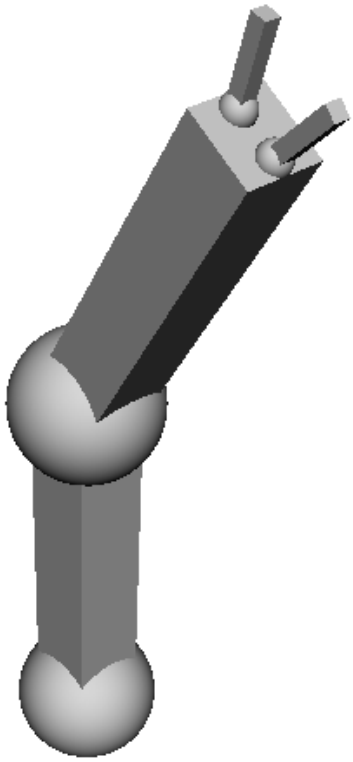


Figure 12.5: A robot arm created using just a sphere and a cube – and hierarchical transformations.

```
  Box
}
EndList
CallList Joint
Translate 0 5 0
Rotate 30 1 0 0
CallList Joint
Translate 0 5 0
{
  Translate 0 0 0.3
  Scale .2 .2 .2
  Rotate 15 1 0 0
  CallList Joint
}
{
  Translate 0 0 -0.3
  Scale .2 .2 .2
  Rotate -15 1 0 0
  CallList Joint
}
```

We do not need to concern ourselves with the syntax of the obscure graphics language used above. The parser simply translates each line into an old school (compatibility mode) OpenGL call. The point is the sequence of transformations. Initially, we define the Joint. The Joint element is then used in four places below. The first instance defines the upper arm, the next defines the lower arm, and the final two instances define the fingers. The transformations given in the script are used to generate transformation matrices that are immediately multiplied onto the current transformation matrix.

Thus, the final transformation depends on both what transformations we use and the ordering. Notice how the transformations closer to the root of the hierarchy are applied first and the transformations closer to the leaves of the hierarchy are applied last. The curly brackets indicate that we push '{' the current transformation or pop it '}'; they are needed when generating the fingers because we need to use different transformations for each finger.

Except for the slightly longer names, this is the same sort of code that we used to generate trees with L-systems. Hopefully, it is quite clear how this sort of script can be used to generate complex branching structures such as a human skeleton. If it is not clear, we encourage the reader to try once more mentally parsing the script.

The very astute reader may have spotted that we are missing a crucial piece! We have just described a way to hierarchically define

the transformation matrices for a complex structure. We know that for each vertex, we need to define how to blend the transformations that influence that vertex. What is missing? Look at the definition of Joint in the script.

As defined, Joint is a ball centered at the origin with a five units long bar (scaled cube) sticking out in the z direction. Bones work in the same way. They are defined to be at the origin pointing in the direction of one of the principal axes. The hierarchical transformation takes them from that initial position to the proper position in the skeleton. If we were to directly transform a vertex from its position in the mesh, the result would be unexpected. Instead, what we need to do is to first transform the vertex to a position that is relative to the bone in its original configuration at the origin and after that we can apply the hierarchical transformation to get the final position of the vertex.

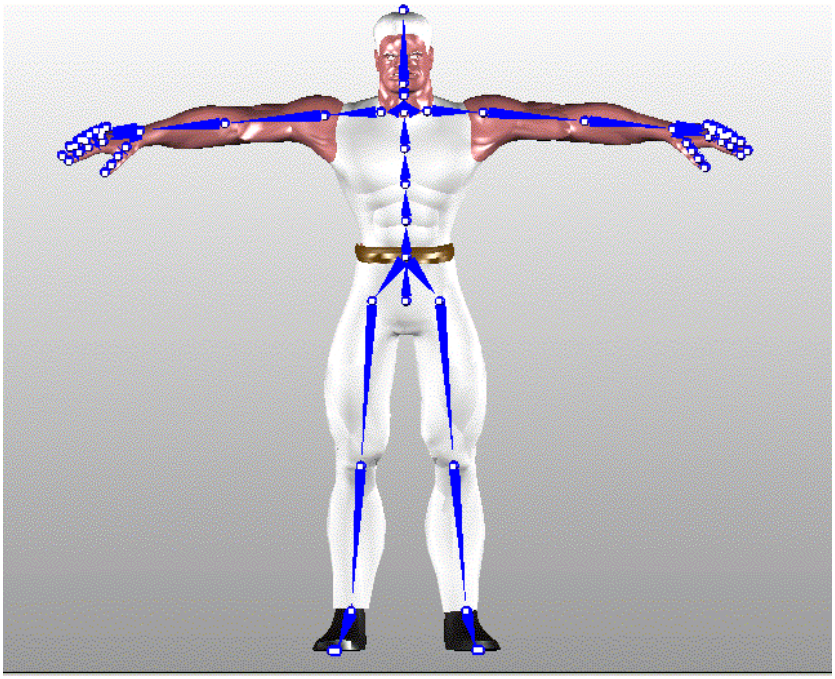


Figure 12.6: Rest pose of 3D model and associated skeleton

To obtain this initial, inverse transformation, we need to define a *rest pose* for the model. An example is shown in Figure 12.6. When doing the rigging, the animator is associating the vertex positions with the bones in the rest pose. Then the inverse of the hierarchical transformation used to place a bone *in the rest pose* is first applied to a vertex. We now have its position relative to the bone in the original

configuration (i.e. at the origin) and from there we can apply a new hierarchical transformation to get the position of the vertex in the animated pose.

With all this in place, here is the final formula for computing the vertex position during animation.

$$\vec{u}(t) = \sum_i w_i \vec{B}_i(t) \vec{M}_i^{-1} \vec{p}$$

where i is an index over all bones, w_i is the weight of bone i with respect to the current vertex. $\vec{B}_i(t)$ is the (hierarchical) transformation of bone i at time t and $\vec{M}_i^{-1}$ is the inverse of the rest pose transformation for bone i. Finally, $\vec{p}$ is the original vertex position.

When implementing this in the vertex shader, we would normally compute the matrices for each bone and for each frame on the CPU and store them in uniform variables. We can premultiply the two matrices and simply store their product (along with a normal matrix). The weights and indices of the bones that actually influence a given vertex are vertex attributes.

What remains to be explained is how we actually obtain the transformation matrices. There are many possibilities ranging from artist generated content over forward or inverse kinematics (i.e. simulation) to motion capture. It would take us too far to go into details with each of these, so we have to be content with simply accepting that we need to somehow have time parametrized transformation for each bone.

12.3 *Flocking: Achieving Emergent Behaviour*

We can animate a 3D model now. It is not much harder to animate a bunch. Now, simply having several animated entities (fish, soldiers, birds, etc.) is not in itself likely to give a very impressive effect. However, if the individual boid (as the reader recalls that we denote such animated entities) is cognizant of the other boids, the effect can be remarkably like a real flock of boids – at least to the casual observer. We say that the high level behavior *emerges* from some (very simple) rules that apply to the individual boid. The principles behind this were studied by Craig Reynolds in 1987.[3]

[3] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987

The most common flocking principles are illustrated in Figure 12.7. The first principle is that boids avoid hitting each other. A simple way to achieve that is by adding a repulsion force (or simply a speed component) to each boid. One can use the weighted sum of vectors from each neighboring boid to the boid under consideration. A good

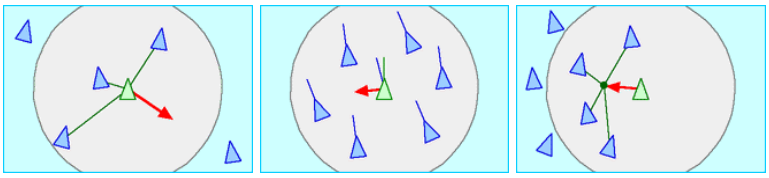


Figure 12.7: Principles that affect a boid. From left to right: collision avoidance, velocity matching, and flock centering.

way of computing the weights is to use a Gaussian falloff function

$$G(\vec{x}) = \exp\left(-\frac{\|\vec{x}\|^2}{\sigma^2}\right)$$

where σ is a constant that is set to some small fraction of the size of the environment. Similarly, we can compute a weighted average of the velocity vectors of neighboring boids and add the resulting velocity to the current velocity. Finally, flock centering pulls us to towards the center of the nearest boids. Instead of finding the center of the nearest boids, we can simply compute the weighted average of vectors towards the other boids. Flock centering needs to have a smaller effect than repulsion, obviously, or we will undo the effect of the latter.

In addition to these effects, we would also add a small random velocity component and we would add a lot of the old velocity to ensure that the boid does not seem to move around randomly like a fly (unless, of course, the boid is supposed to be just that).

A problem that is quickly encountered is that the boids tend to speed up or slow down when applying this machinery. Taming their velocity can be quite challenging, and a better idea seems to be to simply normalize their velocity. Thus, they seem to move at constant speed but change direction as needed.

However, we also need to take walls into account. The repulsion term used for other birds can also be used for walls. Instead of the distance to another boid, we just use the distance to the wall when computing its contribution. Unfortunately, a boid that is swimming directly towards a wall and then turns 180° will appear to instantaneously change direction. If we normalize the speed vector, the rotational speed can still be arbitrary. However, we can limit the rate of turning of a boid if the artifact is objectionable.

From an implementation point of view, it is not quite banal to implement boids. One strategy is to store the state of all boids in a 1D texture. For each texel, we need to know just the position and the velocity. We can then update the boids position and velocity using the principles described above and output a new 1D texture using FBOs.

If our boids move in 3D, we would associate a complete frame of

reference with the boid, i.e. three mutually perpendicular vectors where one is in the direction of velocity and the two others align with the symmetry planes of our boid. If we simply have a school of fish swimming in 2D as in our project, we need only store the velocity. From the scene, we know the "up" vector, $\vec{u}$, and then we compute the final vector in our frame as the cross product of up and velocity. We can construct a rotation matrix by inserting these three vectors as its columns:

$$\vec{M} = [\vec{v} \;\; \vec{u} \times \vec{v} \;\; \vec{u}] \, .$$

Multiplying our object space vertex positions with $\vec{M}$ gives us the rotation of the boid so that it is oriented according to the velocity. Above, we have assumed that the fish is oriented such that its object coordinate x axis corresponds to the way the boid is pointing and that the z axis corresponds to the up direction. This is not guaranteed, so the reader may need to adapt this method. Also, we still need to translate it by adding the position to each vertex.

12.4 Learning More

Clearly, we can combine pose interpolation and bones based animation, and, in fact, this can alleviate some of the limitations of both methods as expounded on by Lewis et al.[4]

The idea behind flocking is due to Craig Reynolds.[5] Boids are really particles. In a later chapter, we will discuss particle systems. The main difference between a system of boids and a particle system is that in the former, we have a complete frame of reference per particle, and, as mentioned, boids know each other.

[4] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000

[5] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987

12.5 Project: A School of Fish

Objective

In this exercise, we will make a very simple animated object, namely a fish. However, one fish would be dull, so we put multiple fish that interact in our basin. This means that the exercise also covers the notion of flocking using a method often known as "boids"

Part 1

In `fish.vert`, the fish drawing vertex shader, insert code which makes the fish move its tail from left to right as a periodic function

of time (see Figure 12.8). At present it is ok that you see just a single fish in the center of the basin. Explain the function you use to move the tail. Hint: The fish is pointing in the direction of -X.

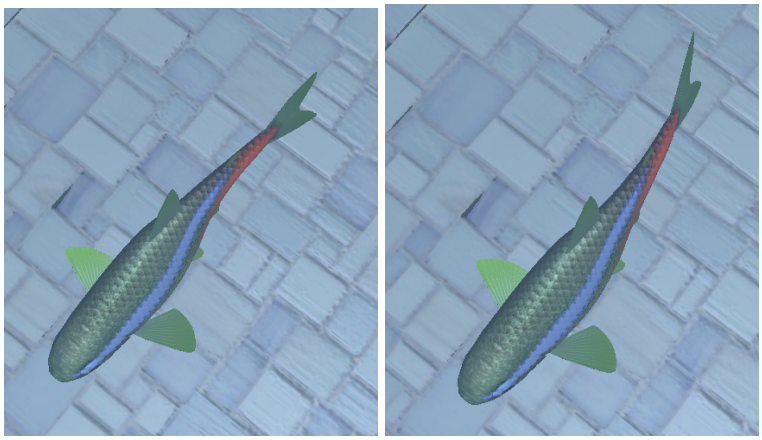


Figure 12.8: An animated fish

Part 2

Next, we need to update the fish position and orientation. This is done in `update_fish.frag`. The principles of how the fish move are simple: The fish is modelled as a particle. However, fish should react to other fish, the user, and the walls of the basin, so the stateless particles used in the previous exercise is not an option. Instead we store the fish in a 1D texture and use a fragment program to update their state. This scheme is often used also for other types of particle systems. Analyze what the C++ code does and explain why we need two textures representing the fish state and not just one. In the shader, `particles_tex` contains information about the sixteen fish. Each pixel in the 1D texture stores the 2D position (XY) and velocity (ZW) of a fish. Each instance of the shader program is meant to update one of the fish, and in the example code, `fish_data` is extracted from `particles_tex` and simply output. Your task is to update the velocity and speed and output these updated values. Now, how do we get the following three terms needed for the velocity?

- A random velocity component
- Alignment of fish swimming nearby
- Avoidance of collision with user, walls, and fish that are too close.

Find out and update the velocity of the particle. Note that a function returning the value of a 2D Gaussian and another function returning

the gradient of a Gaussian have been provided. After updating, you want to renormalize the velocity so that the fish swim at constant speed. Update the position of the particle. Finally, revisit `fish.vert` and move and rotate the fish according to the position and velocity stored in the particle map. Hint: Your speed vector combined with a perpendicular vector can be used as a basis for the xy part of the fish vertices. Experiment with the parameters controlling the velocity terms for the fish motion. Discuss results of the above experiments.

Deliverables

Source code for all parts, answers to all questions (underlined), and screen shots illustrating the two parts.

13
Particle Systems

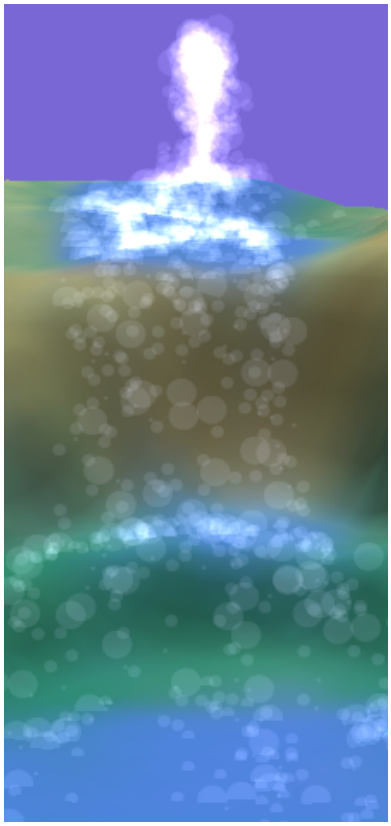


Figure 13.1: A simple fountain cum waterfall implemented as a particle system. It is a simple particle system with just a few thousand particles simulated on the CPU.

Particle Systems are used in computer graphics for a wide variety of effects where the commonality is that we are trying to render something that does not have a firm shape – sometimes called an *amorphous phenomenon*. Amorphous phenomena range from fire over foamy water (as shown in Figure 13.1) to swarms of bugs.

In fact, we have already considered swarms of boids in the chapter on animation, but an important difference is that particles are usually not aware of each other. This means that we can handle many more particles than boids. Also, it is very straight forward to implement particle systems on the GPU.

This chapter begins with a discussion of billboards which are sometimes used to show complex objects reduced to images, and particles can be seen as an ensemble of billboards. After that, we will consider various ways to use particles ranging from the simplest possible case where the particles are stationary and used to represent an object. Later, we will consider stateless particles which, in our humble opinion, is an underrated use case, which is useful if we want to model something like a fountain where the trajectory of each particle is given from its initial condition. Next, we consider particles that *do* have state and can, therefore, be used to model a wider range of phenomena. In that context, we also need to consider how to integrate the particle state.

Finally, we will discuss issues pertaining to particle rendering.

13.1 Billboards

A particle is just a point, and in the next section, we will discuss how points can be used as an object representation. However, we can also use a single point to represent an object. In fact, we can do this in several ways. An obvious strategy is to copy an instance of an object to the place where the point is. We have previously dis-

cussed instancing, and in fact this method has already been used in conjunction with boids. Using an effect like this, we could make a fountain of Stanford bunnies or whatever we like. However, usually, we restrict ourselves and associate only a single quadrilateral with each point. If the quadrilateral is textured then we are basically using sprite rendering or *billboards*.

A billboard can be rendered in several ways as discussed in Real-Time Rendering.[1] If the billboard is used as an impostor (i.e. instead of drawing a complex geometry) for, say, a tree, we would normally constrain the billboard to rotate around the vertical axis given by, $\vec{U}$, the up vector. Equally importantly, we would like the billboard to face the viewer. Knowing that the vector towards the eye, $\vec{V}$, is perpendicular to the billboard, we simply need to take the cross product

$$\vec{B} = \vec{U} \times \vec{V}$$

to obtain $\vec{B}$. Now, $\vec{B}$ and $\vec{U}$ span the plane of our billboard, and we can easily generate the four corners and then two triangles in a geometry shader to produce the geometry of the billboard. This type of billboard is often called an *axial* or axis aligned billboard, and an example is shown in Figure 13.2.

[1] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 2008

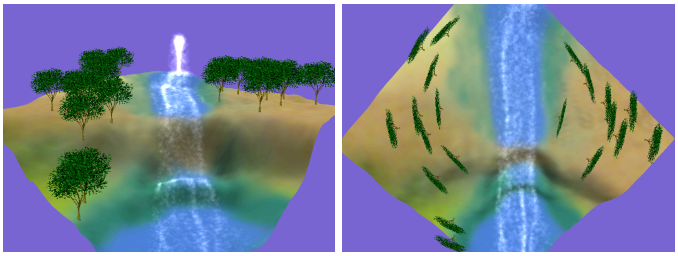


Figure 13.2: A particle effect combined with billboard rendering of trees. billboards are axis aligned: they are rotated around the up axis to face the viewer. However, when the scene is seen from above, the illusion is not impressive.

It is clear from the image that we generally need to include an alpha channel in the billboard texture and use that channel to mask out the parts of the billboard which are supposed be transparent.

Apart from axial billboards, we also have *view plane oriented billboards* which are parallel with the $\vec{X}\vec{Y}$ plane in view coordinates. *View oriented billboards* are simply perpendicular to the view vector.

Below, we will discuss rendering of particle systems. In that case, what we really do is to attach a billboard to each particle. Of course, the billboard in this case is usually simple - we might use just a color instead of texture. One other difference between billboard and particle rendering is that we normally render particles with alpha blending instead of alpha testing: an important aspect of particle rendering

is that the visual impression is conveyed by many particles in combination whereas a billboard is usually supposed to have an effect by itself.

13.2 Particles as an Object Representation

Starting around the year 2000 and for a few years, the use of points or *surfels* as a 3D object representation attracted a lot of research interest. One reason is probably that a point is a very simple primitive which represents both geometry (given a position and normal) and also texture (simply associate a color or a small texture patch with the point). Another good reason for using points as an object representation is that point clouds is precisely what we get out of 3D scanners. Usually points were rendered as small circles or squares, sometimes called point *splats*.

Ultimately, interest waned as it became clear that point rendering is less effective when it comes to rendering smooth surfaces. A large flat region can easily be represented with a few polygons and probably takes longer to render using point primitives. Especially since it is not much faster to render a point than a single triangle – often the opposite. When we zoom in, a point cloud needs to be very dense or we will have to use a very big splat size to avoid holes or gaps between the points. All this does not mean that point cloud rendering is useless. Many devices (e.g. depth cameras) capture point clouds, and it is often very interesting for researchers (or developers) to be able to render the raw point clouds.

In the following, we will consider a (very particular) case where points have a strong merit independent from how the point cloud is generated or acquired. Say, you want to render an extremely complicated surface such as a tree with a lot of foliage. None of the methods typically employed to reduce triangle mesh complexity work well on such an object. However, we can point sample the surface of the tree and thus obtain a point set. If each point has an associated position, normal, and color, we can render the tree simply by rendering this collection of points. Unless the point cloud is sparse, it will probably take longer to render the cloud than the original polygonal mesh, but what if we show the tree at a distance? In that case we need much fewer points: in general, we probably want to have a number of points such that the projection of the point cloud has more or less constant density. In practice, this means that the number of points that we use should be inversely proportional to the square of the distance between the camera and the object.

At first, it might seem difficult to select the points that we use

Figure 13.3: In this scene, the near field trees are rendered using polygons. Trees that are farther away are rendered using point clouds, and the very distant trees with far fewer points than those that are closer.

for rendering. If any sort of computation is required, it might kill the efficiency of the method. Fortunately, all we need to do is to randomize the order of the points. If the points are in truly random order, any fraction of the cloud will yield a reasonable approximation at a great enough distance. We simply need to render just the first n points where n is often much less than the total number of points. Thus, a point cloud is really a continuous LoD impostor. We upload the entire point cloud to the GPU but render only the fraction of the points needed given the distance to the object. The biggest limitation is that when the point cloud gets close, we need to switch back to polygon rendering. Figure 13.3 illustrates the method.

13.3 *Stateless Particles*

While the preceding section shows that we can use particles for static objects, the biggest advantages seem to arise when we consider dynamic phenomena. In principle, our particles can be governed by whatever principles we might choose, but often Newton's laws seem to lead to the desired result. Given this choice, we consider a particle to be a small entity with a given weight, position and velocity.

If the only force that acts on the particle is gravity, and if we know its initial position, $\vec{p}_0$, and velocity, $\vec{v}_0$, we can compute its position at any point in time by analytical integration, i.e.

$$\vec{p}(t) = \frac{1}{2}\vec{a}t^2 + \vec{v}_0 t + \vec{p}_0$$

where $\vec{a}$ is the acceleration due to gravity and t is time. Of course, this assumes that the particle does not hit something or that when it does hit something we simply retire the particle.

For fountains or waterfalls this type of simulation is likely to be sufficient, and the advantage is that we can compute the position of

Figure 13.4: This figure shows a stateless particle system. Each particle is integrated from initial conditions to its current position each frame. Thus, there is no need to store current particle state.

the particle directly in the vertex shader with no need for a separate pass to update the particle state. An example of a stateless particle system is shown in Figure 13.4

13.4 Particles with State

For more complicated simulations and in particular if we wish that our particle simulation should respond to user input, we do need to store state. We often assume that particles have the same weight. In that case, we only need to store its current position and velocity, and simple Euler integration looks as follows

$$\vec{v}(t + \Delta t) = (1 - d)\vec{v}(t) + \Delta t\, \vec{a}$$

$$\vec{p}(t + \Delta t) = \vec{p}(t) + \vec{v}(t)\, \Delta t$$

where $d \approx 0.01$ is the drag coefficient which ensures that particles tend to loose energy over time.

Since we are using stateful particles, we must be planning for something to happen to the particles. For many applications, it would be great if the particles were aware of each other (like in the case of boids) but this would require us to find the particles close to a given particle or to simply compute the interactions between all pairs of particles. Unfortunately, both are too slow if we want hundreds of thousands of particles.

On the other hand, it is certainly feasible to have the particles interact with a static environment: a plane, polygon, sphere, any algebraic surface or a height map are all good examples of objects against which we can collide our particles. Assuming we have found a collision point, we compute the collision response in much the same way that we would reflect a light ray. Assuming the surface has a normal $\vec{N}$, we compute the normal component of the velocity

$$\vec{v}_N = \vec{N}(\vec{N} \cdot \vec{v}) \ ,$$

and the tangential component

$$\vec{v}_T = \vec{v} - \vec{v}_N \ .$$

The velocity after the impact is now

$$\vec{v} = (1 - f)\vec{v}_T - r\vec{v}_N \ ,$$

where f is friction and r is restitution. Thus f slows down the tangential velocity while r determines how much the particle bounces back from the surface. In general f should be a very small, positive number and r between 0 and 1. If it is close to 1, particles become bouncy, and if r is close to zero they act more like clay balls.

We need to store the velocity and position of each particle on the GPU. A common choice is to allocate two 2D textures with three floating point channels each. We store position in one texture and velocity in another. We update the particles by rendering a quadrilateral to a viewport with the same pixel dimensions as the textures. This rendering invokes a shader program that takes one set of textures as input and outputs a new texture to an FBO. Like in the chapter on water simulation, it is common to use a ping pong scheme where we read from one texture and output to another and then switch their roles.

When the particles need to be rendered, we would typically use the point primitive. When drawing each point, we give it a 2D position in the texture image that stores our particles as attribute. In the vertex shader, we now look up the 3D position and velocity of the particle in their respective textures and use that for rendering.

Note that we do not actually need to use two textures. It would be perfectly feasible to use a single texture or even a 1D texture to store the particles. Mentally, it is just slightly easier to come to terms with the use of a texture to store the particle information if there is a one-to-one correspondence between the pixel coordinates and the place in the texture where the point is stored. Going further, with recent versions of OpenGL, we no longer need to use textures to store the point data. It is possible to use transform feedback to capture the output from the vertex shader in a new vertex buffer.

13.4.1 Verlet Integration

Euler integration is not terribly stable, and one idea that has been used very often in particle simulation is a method known as Verlet integration where we get rid of the velocity variable. This is easy since

$$\vec{v}(t) \approx \frac{\vec{p}(t) - \vec{p}(t-1)}{\Delta t} + \vec{a}\Delta t \ .$$

Plugging into the formula for position, we get

$$\vec{p}(t+1) = \vec{p}(t) + \vec{v}(t)\Delta t = 2\vec{p}(t) - \vec{p}(t-1) + \vec{a}\Delta t^2 \ ,$$

or if we allow for a bit of drag:

$$\vec{p}(t+1) = (2-d)\vec{p}(t) - \vec{p}(t-1) + \vec{a}\Delta t^2 \ . \tag{13.1}$$

In exchange for the added stability and loosing the velocity, we need to store one more time step of position data. A practical consequence is that instead of a ping pong scheme, we now need to use three buffers and rotate between the current and two previous states.

13.4.2 *Particle Emission and Recycling*

Clearly, if we start all particles at the same point in time from the same point in space and with the same velocity, we get the appearance of a single particle. In fact, we want particles to be emitted from a source position with a slight random offset and with a velocity that is also given a slight random offset.

The particles generally stay in the scene only for a limited time. Particles that exit the scene should usually be recycled and reintroduced by being given a new start position and start velocity. Knowing roughly how long the particles are in the scene, we can divide this time by the time it takes to render a frame. This gives us a particles expected life time in number of frames. Dividing the total number of particles with this number of frames, we know how many particles to emit each frame. Once all particles are emitted, we simply allow them to be recycled and reemitted.

13.5 *Particle Rendering*

A classical way of rendering particles is to simply output a point primitive. GL can automatically generate a quadrilateral which can be used to render a point sprite. However, in modern OpenGL, we would normally use the geometry shader to convert a point to the desired geometry.

Typically, we would like to render the particle as a quadrilateral that is stretched in the direction of the velocity vector. This is not because a water drop (which the particle might well represent) becomes extremely elongated; instead stretching the visual representation in the particle is a way to introduce motion blurring of the particle.

Thus, we would often render the particle as a line segment stretching from its position in the previous frame to the position in the current frame. The line segment is then turned into a quadrilateral by finding the line which is perpendicular to both the view direction and the velocity vector. We then create four points: a pair of points at the previous position and a pair at the present position. Both pairs are offset slightly along this line. The details are left as an exercise to the reader.

It is important to note that each particles in the case of fire or water spray typically should be rendered with only very low alpha value, say, 0.2, and then additively blended into the frame. Alpha blending should be set up so that the incoming fragment is weighted with its own alpha value and the existing pixel value is weighted by one. Thus, only when many particles are rendered do we see the effect clearly, but it is this aggregate effect that usually works well.

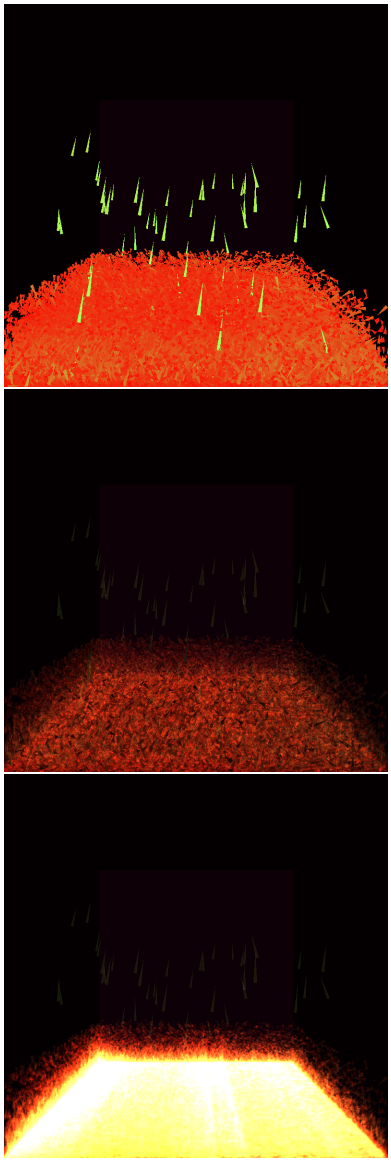


Figure 13.5: Top: particles rendered using the geometry shader to convert each particle into a small triangular arrow of size, direction, and color corresponding to its velocity. Middle: the same particles with blending and depth testing enabled. Bottom: the particles rendered as usual where blending is enabled and the depth test is disabled.

Where the particles cluster we get saturation, and where they don't, they just make the image slightly hazy.

One thing we need to bear in mind is that the particles must be rendered using depth buffer updates switched off. Assume the depth buffer is updated for each particle we draw and bear in mind that we typically do not depth sort the particles. The first particle that covers a given pixel may be the one closest to the camera. In this case, the depth test (which we need to keep active) will reject all subsequent particle fragments (see Figure 13.5 middle). Consequently, this particular pixel receives a contribution from just one particle. On the other hand, if depth buffer updates are switched off, we receive contributions from all particles which are visible with respect to the geometry that was in the frame buffer originally. It is now also clear, that we need to render our particle effect last. Since the depth buffer is not updated, whatever we draw after the particle system will cover it if it is visible with respect to the depth buffer as it looked before we started drawing particles.

13.6 Learning More

Early work on particle simulation was carried out by William Reeves.[2] His methods were used in the movie "Start Trek II: Wrath of Khan". The particular method of Verlet integration is investigated by Thomas Jakobsen.[3] His paper, while easy to find online, does not appear to have been published (except as a Gamasutra article). In fact, particle systems is one of these topics in computer graphics where informal documents that are put online appear to be the best source of information.

[2] William T Reeves. Particle systems: A technique for modeling a class of fuzzy objects. In *ACM SIGGRAPH Computer Graphics*, volume 17, pages 359–375. ACM, 1983

[3] Thomas Jakobsen. Advanced character physics. (apparently not published. Can be found online)

13.7 Project: A Stateless Particle Fountain

Objective

The goal of this exercise is to implement a simple, stateless particle system on the GPU. Stateless means that the position of the particle at any point in time depends only on the initial position and velocity of the particle.

Part 1

In the `particles.vert` vertex shader, you should implement simple Newtonian physics. Each particle has an initial position, a velocity

and a time tag. Compute its position at time `t` based on the initial position, initial velocity, and known Earth gravitation. This position is then transformed and output to `gl_Position`. To see the particles in this part, you need to temporarily disable the geometry shader `particles.geom`. The particles are stateless (we do not store an updated velocity and position). What could be gained by giving them state? Could the same result be obtained otherwise?

Part 2

Particles are simulated as point entities. However, it looks more realistic if the particles are motion blurred by drawing them as quads extended in the velocity direction. Fortunately, we can use the geometry shader to convert a particle to a quadrilateral. In the vertex shader, compute both the position of the vertex and the old position. Pass both to the geometry shader (which you should reenable) and draw a stretched quadrilateral in the plane perpendicular to the direction towards the eye and with the long side parallel to the vector from the position to the old position (both in eye coordinates).

Explain why the layout instructions in the geometry shader are necessary and also what they do. The particles are additively blended into the framebuffer. Explain the procedure, especially issues pertaining to how the depth buffer is used during the particle rendering. Discuss how foam could be added at the point where the jet meets the water surface.

Deliverables

Source code for all parts, answers to all questions, and screen shots illustrating the two parts.

13.8 Project: Cast the Die

Objective

The objective is to write a smartphone app that combines computer graphics using OpenGL ES with the sensors of the phone. You are going to work on a Nokia N9 mobile phone (which you may borrow from us) using Qt Creator.

Part 1

Your first task is to make sure you can talk to the phone via USB, open the project (available on CN/resource) on the PC, compile and

send the app to the phone. These steps will have been illustrated during the lecture.

Part 2

Change the program such that the cube is animated like a particle in a small box affected only by the acceleration measured by the accelerometer. Thus, when the phone is rotated, the particle will move. Describe the physics simulation used to compute the particle's motion. The application should work almost like a small box with a cube inside. In what ways does it differ? Answer the following questions How is OpenGL ES different from regular OpenGL? What type of GPU architecture is used in the N9?

Part 3

Do one or more of the following extensions:

- Make the cube move as the user swipes the display.
- Do Phong based shading of the cube.
- Let the light sensor determine the intensity of the illumination of the cube.
- Make the ball roll as it moves.

Deliverables

Source code you wrote for the app, answers to all questions, and screen shots illustrating Parts 2 & 3.

14
Bibliography

[1] Frederik P. Aalund. A comparative study of screen-space ambient occlusion methods. B.Sc. Thesis, Technical University of Denmark, February 2013.

[2] Henrik Aanæs. *Lecture Notes on Camera Geometry*. DTU Informatics, 2009.

[3] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 2008.

[4] J Andreas Baerentzen and Henrik Aanaes. Signed distance computation using the angle weighted pseudonormal. *Visualization and Computer Graphics, IEEE Transactions on*, 11(3):243–253, 2005.

[5] J Andreas Bærentzen, Jens Gravesen, François Anton, and Henrik Aanæs. *Guide to computational geometry processing: foundations, algorithms, and methods*. Springer Science & Business Media, 2012.

[6] Jakob Andreas Bærentzen, Marek Krzysztof Misztal, and K Wełnicka. Converting skeletal structures to quad dominant meshes. *Computers & Graphics*, 36(5):555–561, 2012.

[7] James F. Blinn. Simulation of wrinkled surfaces. *ACM Siggraph Computer Graphics*, 12(3):286–292, 1978.

[8] Robert Bridson and Matthias Müller-Fischer. *Fluid simulation: SIGGRAPH 2007 course notes*. ACM, 2007.

[9] J. M. Carstensen. *Image analysis, vision, and computer graphics*. Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2001.

[10] Paul W de Bruin, FM Vos, Frits H Post, SF Frisken-Gibson, and Albert M Vossepoel. Improving triangle mesh quality with

surfacenets. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2000*, pages 804–813. Springer, 2000.

[11] Rouslan Dimitrov. Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007.

[12] David S Ebert, F Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 3rd edition, 2002.

[13] David Eppstein, Michael T Goodrich, Ethan Kim, and Rasmus Tamstorf. Motorcycle graphs: canonical quad mesh partitioning. In *Computer Graphics Forum*, volume 27, pages 1477–1486. Wiley Online Library, 2008.

[14] Jeppe Revall Frisvad. Building an orthonormal basis from a 3d unit vector without normalization. *Journal of Graphics Tools*, 16(3):151–159, August 2012.

[15] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 447–452, New York, NY, USA, 1998. ACM.

[16] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[17] Markus Hadwiger, Joe M Kniss, Christof Rezk-Salama, Daniel Weiskopf, and Klaus Engel. *Real-time volume graphics*. AK Peters, Ltd., 2006.

[18] Thomas Jakobsen. Advanced character physics. (apparently not published. Can be found online).

[19] Mark Jones, J Andreas Baerentzen, and Milos Sramek. 3d distance fields: A survey of techniques and applications. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):581–599, 2006.

[20] Donald E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, volume 2. Addison-Wesley, 1997.

[21] Donald E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1998.

[22] John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and

skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000.

[23] Lee Markosian. *Art-based Modeling and Rendering for Computer Graphics*. PhD thesis, Providence, RI, USA, 2000. AAI9987803.

[24] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, pages 23–24, 2005.

[25] Morgan McGuire, Brian Osman, Michael Bukowski, and Padraic Hennessy. The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 25–32. ACM, 2011.

[26] G S Miller and C R Hoffman. Illumination and reflection maps: Simulated objects in real environments. In *ACM SIGGRAPH 84 Course Notes for Advanced Computer Graphics Animation*, 1984.

[27] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, pages 97–121. ACM, 2007.

[28] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[29] Ken Perlin and Eric M Hoffert. Hypertexture. *Computer Graphics*, 23(3):253–262, 1989.

[30] Matt Pharr and Simon Green. Ambient occlusion. In *GPU Gems*, chapter 17. Addison-Wesley, 2004.

[31] A J Preetham, P Shirley, and B Smits. A practical analytical model for daylight. In *Proceedings of ACM SIGGRAPH 1999*, pages 91–100, 1999.

[32] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomir Mech. L-systems: from the theory to visual models of plants. In *Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences*, volume 3, pages 1–32. Citeseer, 1996.

[33] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S Hanan, F David Fracchia, Deborah R Fowler, Martin JM de Boer, and Lynn Mercer. *The Algorithmic Beauty of Plants*. Number 6. Springer-Verlag, 1990.

[34] William T Reeves. Particle systems: A technique for modeling a class of fuzzy objects. In *ACM SIGGRAPH Computer Graphics*, volume 17, pages 359–375. ACM, 1983.

[35] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[36] Kirk Riley, David S Ebert, Martin Kraus, Jerry Tessendorf, and Charles Hansen. Efficient rendering of atmospheric phenomena. In *Proceedings of the EGSR 2004*, pages 375–386, 2004.

[37] Bahaa E. A. Saleh and Malvin Carl Teich. *Fundamentals of Photonics*. John Wiley & Sons, Hoboken, New Jersey, second edition, 2007.

[38] Marc Stamminger and George Drettakis. Perspective shadow maps. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 557–562. ACM, 2002.

[39] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, pages 270–274. ACM, 1978.

[40] Fan Zhang, Hanqiu Sun, and Oskari Nyman. Parallel-split shadow maps on programmable gpus. *GPU Gems*, 3:203–237, 2007.