Downloaded from orbit.dtu.dk on: Dec 17, 2017

Technical University of Denmark

Lecture Notes on Real-Time Graphics

Bærentzen, Jakob Andreas

Publication date:
2010

Document Version
Early version, also known as pre-print

Link back to DTU Orbit

Citation (APA):
Bærentzen, J. A. (2010). Lecture Notes on Real-Time Graphics.

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Lecture Notes on Real-Time Graphics

J. Andreas Bærentzen
jab@imm.dtu.dk
DTU Informatics

October 23, 2009

Contents

1 Introduction **3**
- 1.1 Note to the Reader 3
- 1.2 Acknowledgements 4

2 Overview of the pipeline **4**

3 Vertex Transformation and Projection **5**
- 3.1 Model Transformation 7
- 3.2 View Transformation 9
- 3.3 Projection 10
  - 3.3.1 Orthographic Projection 11
  - 3.3.2 Perspective Projection 12
  - 3.3.3 Clipping 13
  - 3.3.4 W divide 14
- 3.4 Viewport Transformation 14

4 Triangle Rasterization and the Framebuffer **14**
- 4.1 Rasterization 15
- 4.2 Interpolation of Attributes 16
- 4.3 Perspective Correct Interpolation 18
  - 4.3.1 The Details of Perspective Correct Interpolation 19
- 4.4 Depth Buffering 22

5 Per Vertex Shading **23**
- 5.1 Phong Illumination Model 25

6 Texture Mapping **27**
- 6.1 Interpolation in Texture Images 29
  - 6.1.1 Anisotropic Texture Interpolation 32

7 Programmable Shading **33**
- 7.1 Vertex and Fragment Shaders 34
- 7.2 Animation 35
- 7.3 Per pixel lighting 35
- 7.4 Deferred Shading and Image Processing 35

8 Efficent Rendering **36**
- 8.1 Triangle Strips 36
- 8.2 Indexed Primitives 38
- 8.3 Retained Mode: Display Lists, Vertex Buffers, and Instances 38

9 Aliasing and Anti-Aliasing **39**

10 Conclusions **41**

1 Introduction

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 two main APIs in use: DirectX and OpenGL. The former is specified by Microsoft and the supported DirectX version is the most useful way of specifying the capabilities of a graphics card. The latter is the most useful API for cross platform development, since OpenGL is the only option on most platforms that do not run some version of Microsoft Windows. Both APIs have bindings for several languages and both have derivative APIs for specialized platforms such as game consoles or mobile platforms.

1.1 Note to the Reader

The goal of this lecture note 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 also emphasize math. In particular, we discuss the coordinate transformations and the interpolation techniques which are used throughout the real-time rendering pipeline.

You do not need to understand graphics programming to understand this text, nor will you learn graphics programming *from* this text. These lecture notes go together with some exercises in a simple programming language such as Matlab and my own tool TOGL which is short for Text OpenGL. Neither Matlab nor TOGL require you to learn C++ or to even use a compiler. However, knowledge of Matlab or TOGL is also not a requirement for understanding this text, but you can learn about TOGL from the documentation [4].

If you do want to dig deeper and learn graphics programming with OpenGL using C or C++, we recommend Angel's book as a place to start [3]. Haines, Akenine-Möller, and Hoffman's book Real-Time Rendering is also an important source which covers almost all aspects of real-time graphics to the point of being nearly an encyclopedia [2].

In summary, this text is a very quick introduction to the principles of real-time graphics. Hopefully, it will whet your appetite for doing computer graphics and give you a solid understanding of the basic principles.

1.2 Acknowledgements

Jeppe E. Revall Frisvad and Marek K. Misztal found numerous spelling mistakes and unclear phrasings, and correcting these issues greatly improved the text.

2 Overview of the pipeline

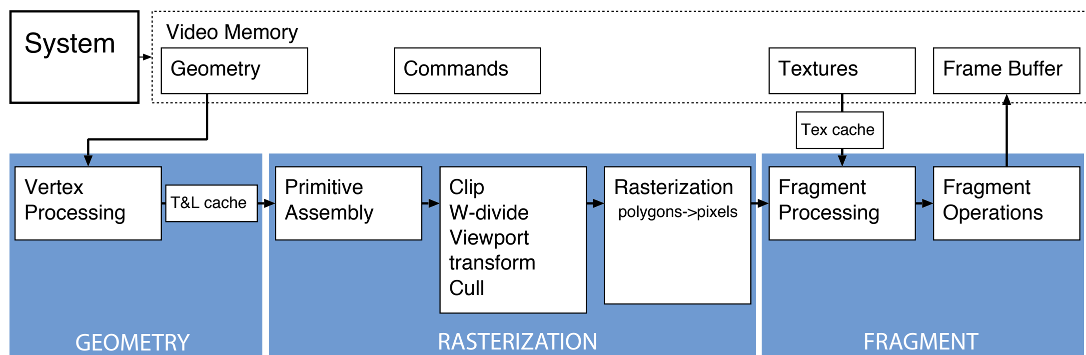

Figure 1: The pipeline in a graphics card. We can roughly divide the pipeline into a geometry part where vertices are transformed, rasterization where triangles are turned into fragments (potential pixels), and a fragment part where we process fragments and finally write them to the framebuffer.

The pipeline of a graphics card is illustrated in Figure 1. The input is geometry in the form of triangles, textures, and graphics commands. The corners of triangles are

denoted *vertices*, and 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, 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 *cull* (i.e. remove from the pipeline) triangles which are invisible, 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. Often, this is simply done by interpolating the color between the vertices of the triangle. After shading, some more fragment operations are performed, and, finally, 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.

3 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].

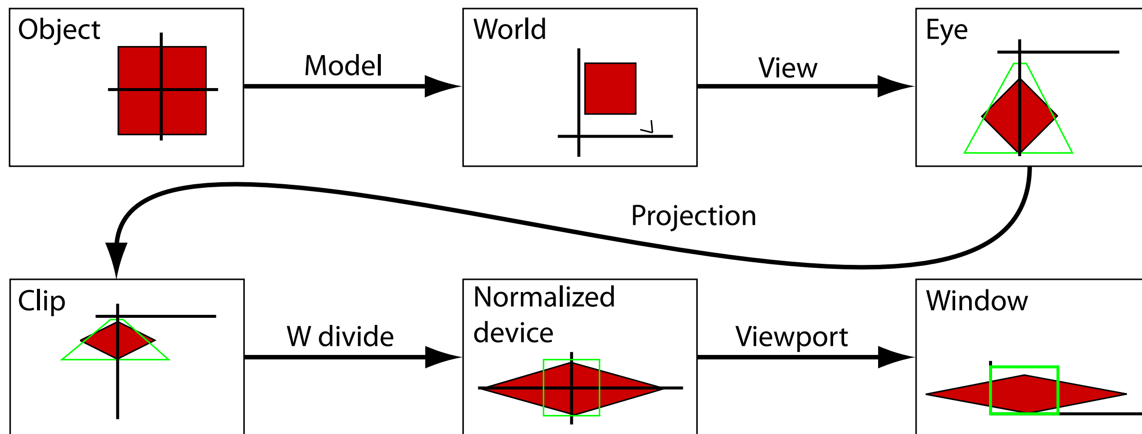

Figure 2: The coordinate systems used in real-time rendering. The words in the boxes denote coordinate systems, and the words over the arrows denote transformations. The red box is the object we render, and the coordinate axes are indicated by heavy black lines. The viewing frustum of the virtual camera 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).

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). 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 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.

In the rest of this section, we will describe this pipeline of transformations in a bit more

detail. Note that the transformations could be done in several ways sometimes, but we follow the procedure described in the specification for the OpenGL API [6].

3.1 Model Transformation

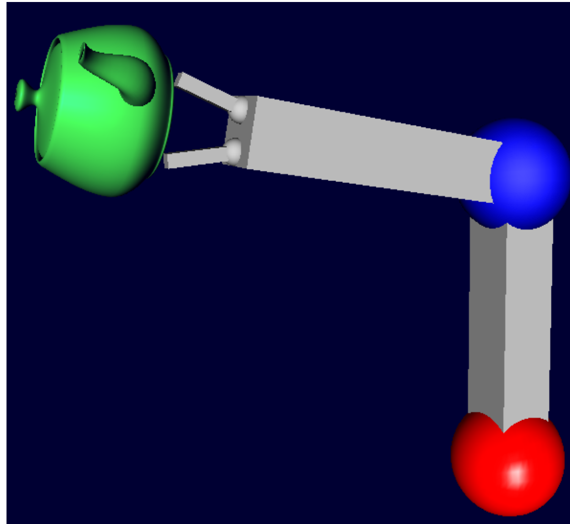

Figure 3: A robot arm created with simple primitives (apart from the teapot). Each primitive has been transformed from object to world coordinates with its own model transformation.

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 material used for rendering. For instance, observe the robot arm "holding" a teapot in Figure 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 [1]. In homogeneous coordinates, the 3×3 rotation matrix, $\mathbf{R}^{3\times3}$ is

simply the upper left 3×3 matrix of a 4×4 matrix:

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

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

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

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

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

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

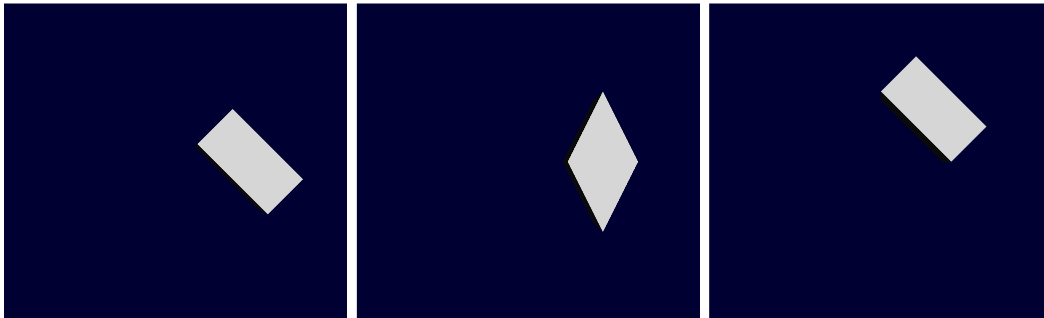

Figure 4: These three images show the significance of the order of transformations. To the left, the box has been scaled, rotated, and translated. In the middle it has been rotated, scaled, and translated. To the right it has been translated, scaled, and rotated. The individual transformations are the 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 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 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.

3.2 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, $\mathbf{e}$, the direction that the camera points in (i.e. line of sight), $\mathbf{d}$, and an up vector, $\mathbf{u}$. The up vector points in the direction whose projection should correspond to the screen Y axis. See Figure 5.

Based on these three vectors, we need to compute the translation and rotation of the camera. The camera translation is simply $-\mathbf{e}$ 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 $\mathbf{c^x}$, $\mathbf{c^y}$, and $\mathbf{c^z}$. The last one is fairly easy. Since the camera looks down the negative Z axis, we have that

$$\mathbf{c^z} = -\mathbf{d} \tag{1}$$

We need to compute a $\mathbf{c^x}$ 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:

$$\mathbf{c^x} = \mathbf{d} \times \mathbf{u} \tag{2}$$

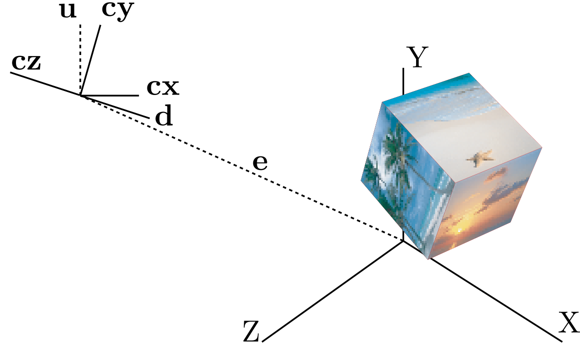

Figure 5: The vectors **d**, **e**, and **u** which are needed to compute the basis (in world coordinates) for the eye coordinate system as well as the basis itself ($\mathbf{c^x}$,$\mathbf{c^y}$,$\mathbf{c^z}$)

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

$$\mathbf{c^y} = \mathbf{c^z} \times \mathbf{c^x} \tag{3}$$

These three vectors are normalized to length 1. Now, we can write down the viewing transformation

$$\mathbf{V} = \begin{bmatrix} {c^x}_x & {c^x}_y & {c^x}_z & 0 \\ {c^y}_x & {c^y}_y & {c^y}_z & 0 \\ {c^z}_x & {c^z}_y & {c^z}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

Note that in the OpenGL API there is only a single modelview transformation matrix which we will denote **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.

3.3 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 6.

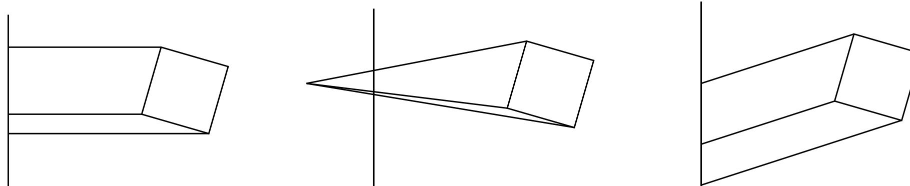

Figure 6: From left to right, this figure illustrates orthographic, perspective, and oblique projections.

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 4.4. 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 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.

3.3.1 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 $\mathbf{p}_e = [r \ t \ -f \ 1]^T$, the point in normalized device coordinates is

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

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 $\mathbf{O}$.

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 7.

After parallel projection, the final step is viewport transformation which is described in Section 3.4.

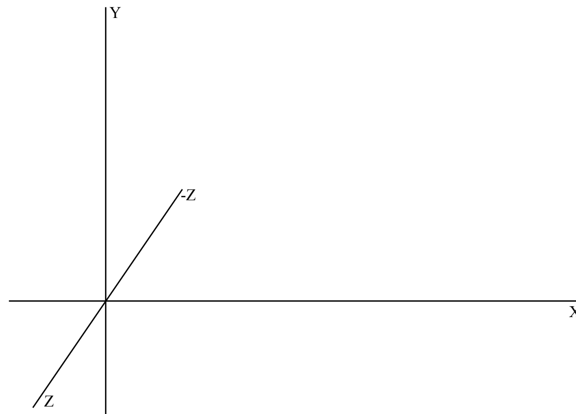

Figure 7: A normal right handed coordinate system. Note that we need to look down the negative Z axis if we want the Y axis to point up and the X axis to point to the right.

3.3.2 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 8.

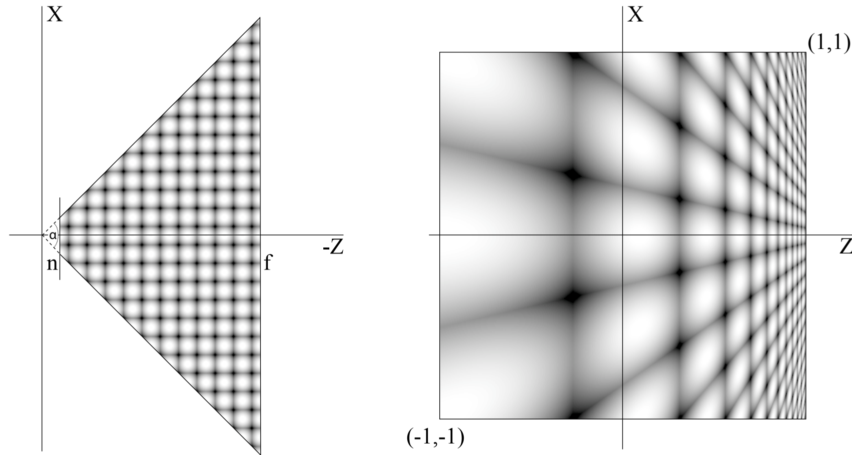

Figure 8: This 2D figure illustrates the perspective projection (including perspective divide). The view frustum is shaped like a pyramid with the top cut off at the near plane n and the bottom cut at the far plane f. The field of view angle, α determines how pointy the frustum is. The perspective projection 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.

In computer graphics, we often use the following matrix for perspective projection

$$\mathbf{P} = \begin{bmatrix} A\cot\frac{\alpha}{2} & 0 & 0 & 0 \\ 0 & \cot\frac{\alpha}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{5}$$

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 8 except for A which is the ratio of the width to the height of the on screen window. Now, we compute clip coordinates by

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

Note that $\mathbf{P}$ 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.

3.3.3 Clipping

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

$$\begin{array}{c} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ -w_c \leq z_c \leq w_c \end{array} \tag{6}$$

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.

3.3.4 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 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.

3.4 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:

$$\mathbf{p}_p = \begin{bmatrix} W\frac{x_n+1}{2} \\ H\frac{y_n+1}{2} \\ \frac{z_n+1}{2} \end{bmatrix} \tag{7}$$

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.

4 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.

4.1 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 not always published and in any case beyond the scope of this introductory text. However, 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. 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.

4.2 Interpolation of Attributes

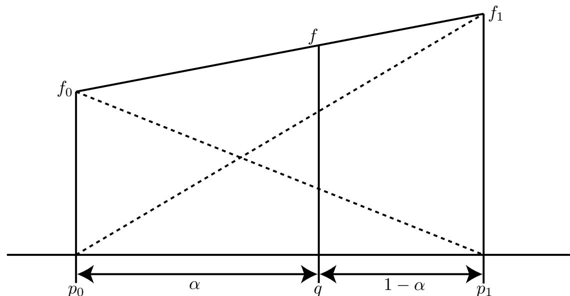

Figure 9: Linear interpolation between points p_0 and p_1 on the real line. The dotted lines show $(1-\alpha)f_0$ and αf_1 whereas the solid line connecting (p_0, f_0) and (p_1, f_1) is the sum $((1-\alpha)f_0 + \alpha f_1)$ and the line on which the interpolated values lie.

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 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

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

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

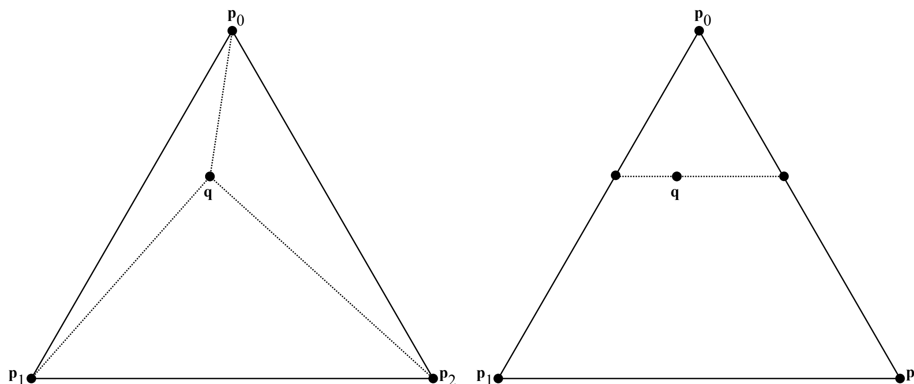

Figure 10: This figure illustrates the triangles and points involved in computing barycentric coordinates on the left. On the right an alternative scheme for linear interpolation where we interpolate using 1D linear interpolation to intermediate points on the same horizontal line and then interpolate to $\mathbf{q}$ along the horizontal line.

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

$$\mathbf{b} = [b_0, b_1, b_2]^T = \frac{1}{A(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)} \left[A(\mathbf{q}, \mathbf{p}_1, \mathbf{p}_2),\ A(\mathbf{p}_0, \mathbf{q}, \mathbf{p}_2),\ A(\mathbf{p}_0, \mathbf{p}_1, \mathbf{q})\right]^T$$

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

$$f_{\mathbf{q}} = b_0 f_0 + b_1 f_1 + b_2 f_2 \tag{8}$$

As we can see from Figure 10, 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 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 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.

4.3 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.

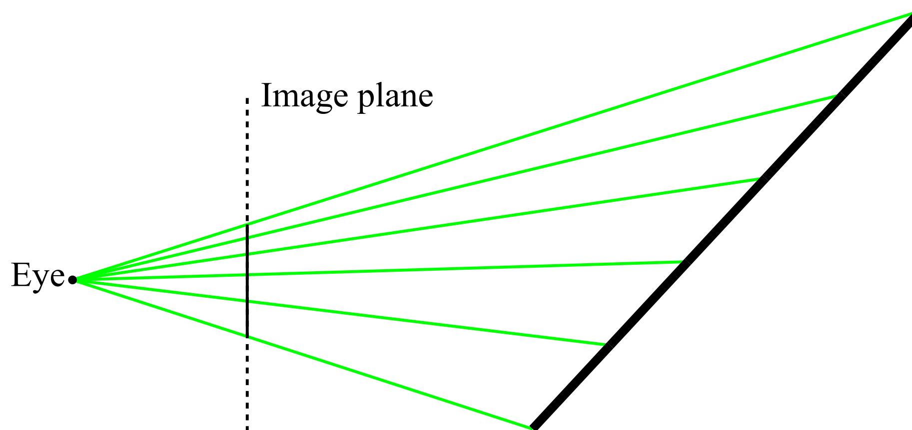

Figure 11: A line in perspective. The line is divided into equal segments, but these equal segments do not correspond to equal segments in the image. Thus, linear interpolation in object coordinates and image coordinates does not produce the same results.

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 19.

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

$$f_{\mathbf{q}} = \frac{b_0 \frac{f_0}{w_0} + b_1 \frac{f_1}{w_1} + b_2 \frac{f_2}{w_2}}{b_0 \frac{1}{w_0} + b_1 \frac{1}{w_1} + b_2 \frac{1}{w_2}} \quad . \tag{9}$$

In the following, we shall see why.

4.3.1 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 12

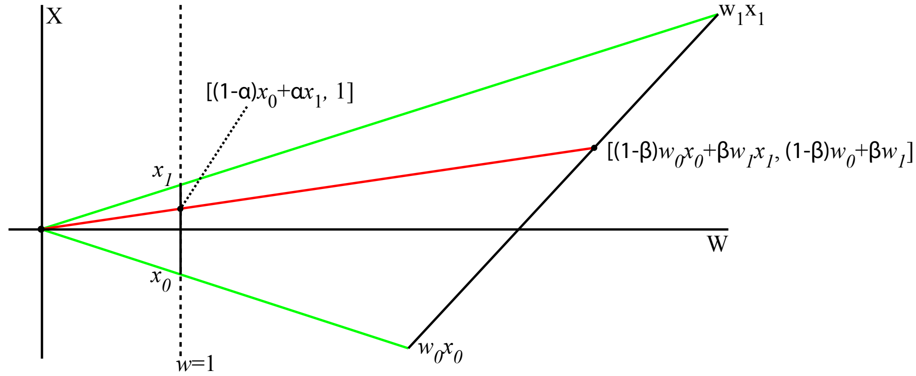

Figure 12: A line in perspective. The line is divided into equal segments, but these equal segments do not correspond to equal segments in the image. Thus, linear interpolation in object coordinates and image coordinates does not produce the same results.

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:

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

The interpolation can also be written

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

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

$$\beta = \frac{\alpha w_0}{w_1 - \alpha(w_1 - w_0)} \tag{10}$$

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 (10),

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

which is straight forward to rewrite to

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

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 (8) to take perspective into account

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

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.

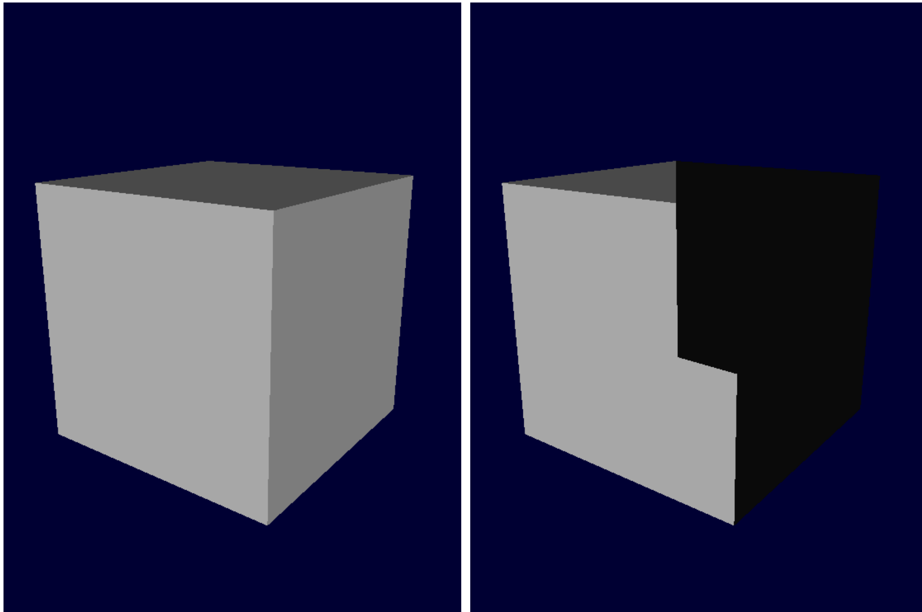

Figure 13: The result of drawing a box with depth test (on the left) and without depth test (on the right).

4.4 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 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.

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 are to draw the model in strict back to front order.

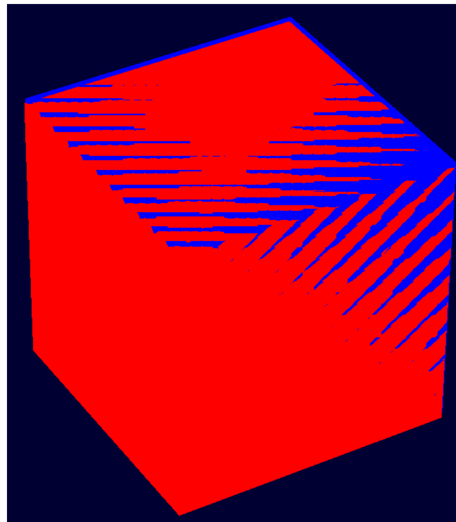

Figure 14: If the near plane is very close to the origin, the depth buffer precision near the rear end of the view volume becomes very poor. The result is depth fighting artefacts. In this case the blue cube is behind the red cube but shows through in several places.

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 8). In fact, it is more scaled the closer to the origin the near plane lies. 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 14.

5 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 (at least in OpenGL) 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*, $\mathbf{n}_o$, into eye coordinates ($\mathbf{n}_e$). 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 $\mathbf{v}_o$ in the tangent plane of the point, we have that

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

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

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

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 $\mathbf{p}_o^1$ and $\mathbf{p}_o^2$ in object coordinates. If the vector from $\mathbf{p}_o^1$ to $\mathbf{p}_o^2$ is perpendicular to the normal, we can express this relationship as follows

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

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

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

which is the same as

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

Thus, we can transform the normal by, $((\mathbf{MV})^{-1})^T$, the transpose of the inverse. This is guaranteed to work if the modelview matrix is non-singular (i.e. has an inverse). 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.

5.1 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 $\mathbf{p}_e = \mathbf{MVp}_o$.
- The normal $\mathbf{n}_e = ((\mathbf{MV})^{-1})^T \mathbf{n}_o$.
- The light source position $\mathbf{pl}_e$.

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

$$\mathbf{v} = -\frac{\mathbf{p}}{\|\mathbf{p}\|} \quad . \tag{11}$$

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

$$\mathbf{l} = -\frac{\mathbf{pl} - \mathbf{p}}{\|\mathbf{pl} - \mathbf{p}\|} \quad . \tag{12}$$

The vectors involved in lighting computation are shown in Figure 15.

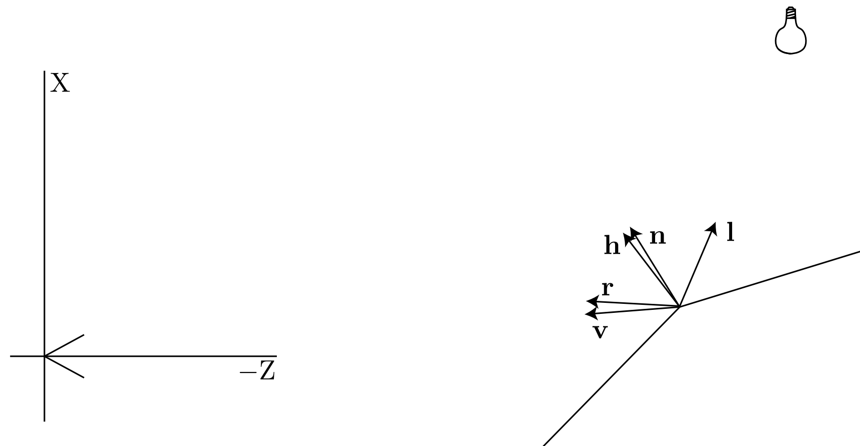

Figure 15: The vectors needed to compute shading according to the Phong and Blinn-Phong illumination model.

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

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

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

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

The specular contribution is

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

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 **h** is defined as the normalized average of the view and light vectors

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

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

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

The 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, **v** and **l** are both constant, and as a consequence so is **h**. It is our understanding that graphics hardware uses Blinn's formulation since setting the view and light vectors constant seems to improve frame rate perceptibly. If we combine the specular, diffuse, and ambient contributions we get the

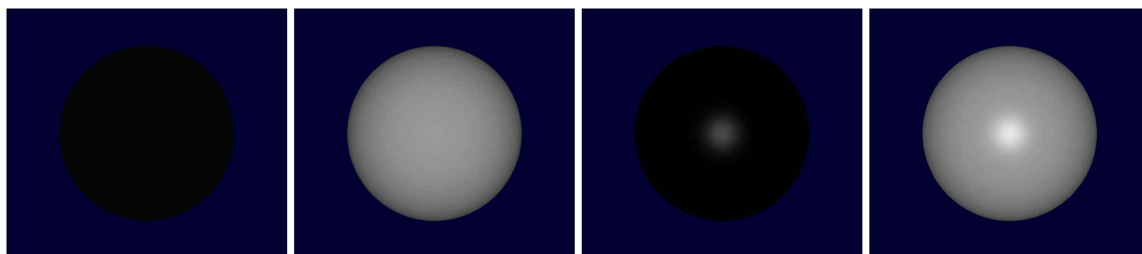

Figure 16: From left to right: The contributions from ambient (almost invisible), diffuse, and specular reflection. On the far right the combination of these lighting contributions.

following equation for computing the color at a vertex

$$L = L_a + L_d + L_s = k_a I_a + k_d(\mathbf{n}\cdot\mathbf{l})I_d + k_s(\mathbf{h}\cdot\mathbf{n})^p I_s \tag{13}$$

Figure 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.

6 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 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.

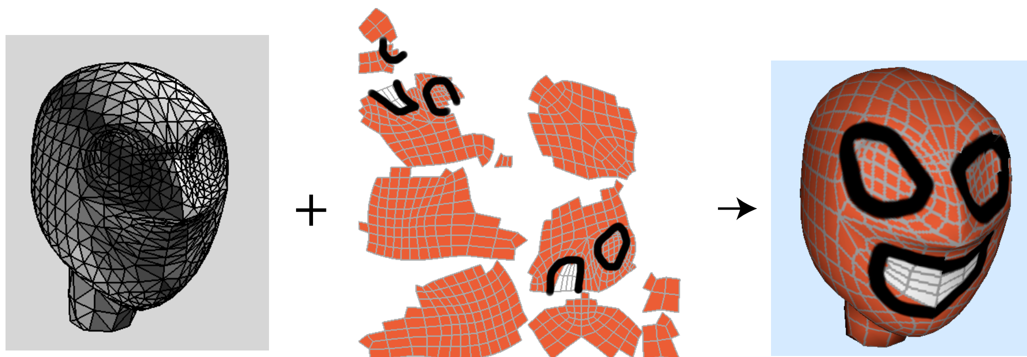

Figure 17: This figure illustrates the principle of texture mapping. Texture stored in a planar image is mapped onto the mesh.

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 more 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 18.

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 19.

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 20. 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

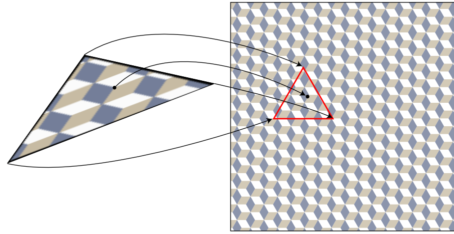

Figure 18: Texture coordinates provide a mapping from the geometric position of a vertex to its position in texture space. When a triangle is drawn, we can interpolate its texture coordinates to a given pixel and hence find the corresponding position in the texture image.

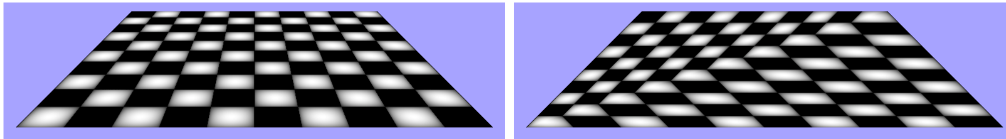

Figure 19: Perspective correct interpolation is important for texture coordinates. On the right we see what happens if the texture coordinates are interpolated linearly in window space.

pixels and 1 for foreground pixels. Alpha testing is the used to remove pixels with value 0 since graphics hardware can filter pixels based on their alpha value and thus cut out the background parts of a texture image as illustrated in Figure 20.

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 17.

We can do many other things with texture - especially with the advent of programmable shading (cf. Section 7).

6.1 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.

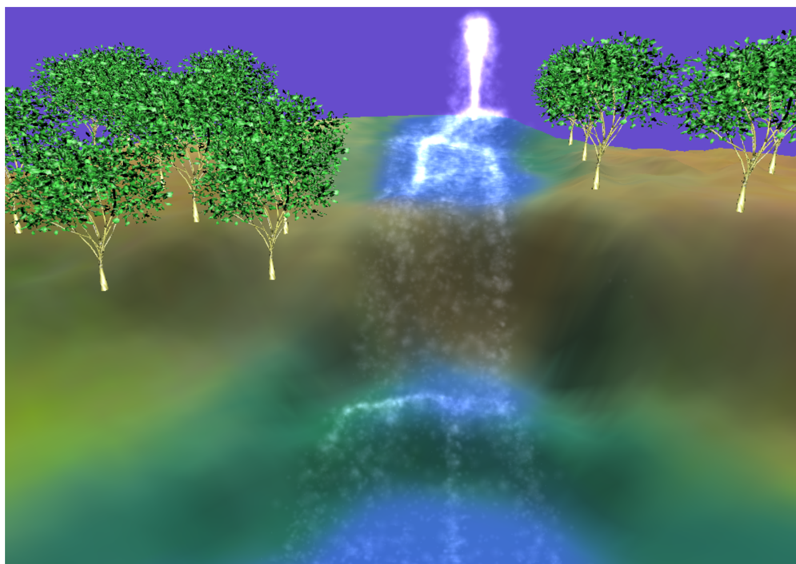

Figure 20: There are no 3D models of trees in this image. Instead a single tree was drawn in an image which is used as billboard. This way of drawing trees is a bit dated. Nowadays one would use many more polygons to define the tree.

This can be fixed through interpolation. GPUs invariably use bilinear interpolation[1] 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:

$$f = (1-\beta)((1-\alpha)f_0 + \alpha f_1) + \beta((1-\alpha)f_2 + \alpha f_3) \tag{14}$$

where f_i are the quantities we interpolate and the weights are α and β. See Figure 21 for an illustration.

It 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 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

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

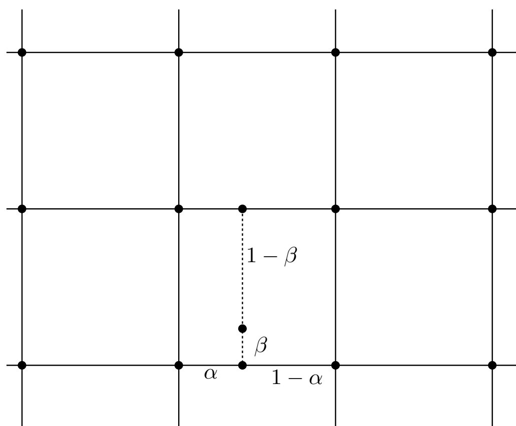

Figure 21: In bilinear interpolation, we interpolate between four data points which lie on a regular grid. It is implemented in terms of three linear interpolations. We first interpolate to two intermediate points and then between these two intermediate points.

pixels.

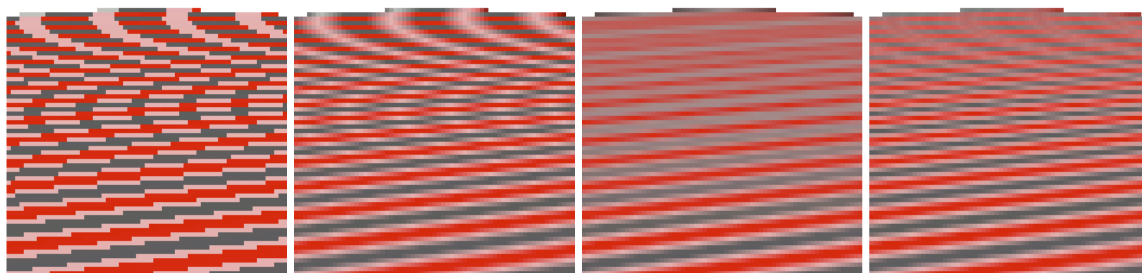

Figure 22: From left to right: Nearest texel, bilinear interpolation, mipmapping, and anisotropic interpolation.

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.

6.1.1 Anisotropic Texture Interpolation

Mipmapping is a very important technique, but it is not perfect. As we see in Figure 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.

In Figure 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 fo the issue in Figure 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

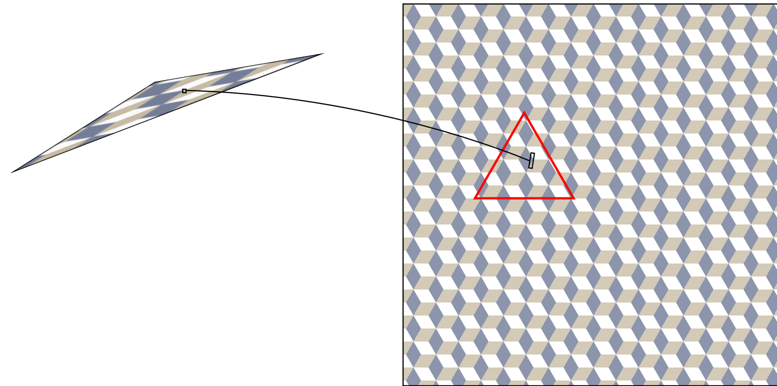

Figure 23: A single square pixel mapped back into texture space becomes a very stretched quadrilateral (four sided polygon).

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 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.

7 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.

7.1 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. In recent graphics cards, 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.

7.2 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.

7.3 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.

7.4 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 24.

8 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. C.f. Figure 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.

8.1 Triangle Strips

Figure 25 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.

Figure 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.

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.

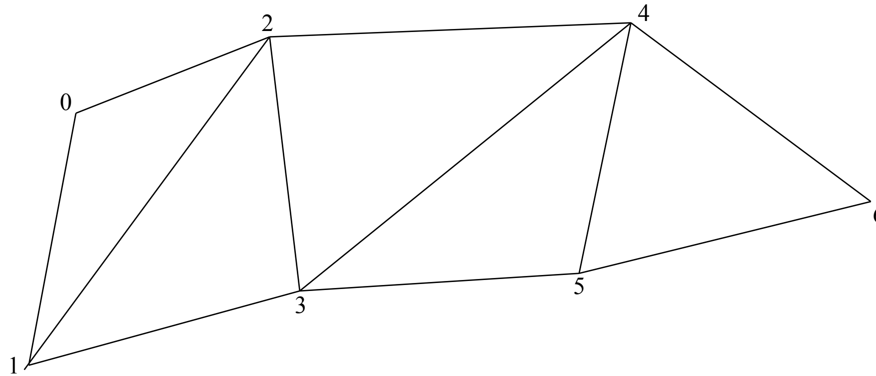

Figure 25: A triangle strip is a sequence of vertices where every new vertex (after the first two vertices) gives rise to a triangle formed by itself and the two preceding vertices.

8.2 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.

8.3 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, and even *deprecated* (subject to removal) in recent versions of the normal OpenGL API.

However, there is an easy fix to the problem, 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. Therefore, display lists are also deprecated. 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.

9 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 sampling 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 26.

Figure 26: The difference between no anti-aliasing on the left and anti-aliasing with 16 samples per pixel on the right.

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 textures. 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.

10 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.

References

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

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

[3] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL (5th Edition).* Addison Wesley, 5 edition, 2008.

[4] J. Andreas Bærentzen. *TOGL: Text OpenGL.* DTU Informatics, 2008.

[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.

[6] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Ver 3.1).* The Khronos Group, March 2009.