Exercises for Essential Mathematics for Games and Interactive Applications, Second Edition

Chapter 1

For the floating-point exercises, we'll often use FP16 as described in the text, rather than the IEEE standard single precision. The desired format will be listed in each exercise.

Note that in several of these exercises, we will require that floating-point values be treated as their bitwise representation, so they can be bitmanipulated. This is not the same as using C-language casting from `float` to `int`. Rather, it involves reading or writing the floating-point storage as an unsigned integer of the same size. A common trick for this is:

```
float myFloat = ???;
unsigned int myFloatAsBits = *((unsigned int*)(&myFloat));
```

This code treats the storage of the `float` as `int` storage and makes it possible to manipulate the bits.

Note further that several of these exercises could be computed using C code and a debugger to view the results as bitwise floating-point values. This is clearly not the point of the exercise (although it is one way to check your final result for correctness); the value of the respondent's answer is to be judged by how well it shows every step of the conversion process or floating-point operation process. This is a chance for the respondent to show how well they know the steps of the operations. Each step in the process should be laid out, when possible, in an easy-to-read bit-table representation similar to those used in the text.

1.1. Convert the following real numbers into single-precision IEEE floats. Also, convert them to `FP16` values:

$$A = 24.5$$
$$B = 48.75$$

1.2. Based on the above result, show the steps and results of the computations listed below in both IEEE single-precision and `FP16`:

$$A + B$$
$$A - B$$
$$A \times B$$

1.3. Without using any built-in floating-point operations (e.g., C standard library or C-style casting), write C-code functions to take an IEEE single-precision floating-point value, represent it as bits, do the following floating-point operation, and return the correct floating-point result. Do not concern yourself with overflow or denormalized cases.

1. "Shifting left", that is, multiply the given floating-point value by a power of two.
2. Absolute value (`fAbs`).
3. Convert to integer (integer cast).

1.4. Write C-code functions to convert nonexceptional (no overflow or denormalized values) IEEE single-precision floating-point values to `FP16` values. Write another function to convert `FP16` values to IEEE single precision. The `FP16` values should be represented as a 16-bit `unsigned short` value in C. Then implement the following for `FP16`:

1. Conversion from IEEE to `FP16` can be inexact, since `FP16` is less precise. Expand the IEEE-to-`FP16` function to return the relative error of representation of the conversion as an IEEE single-precision value.
2. Reimplement the "Shifting left," absolute value, and conversion to integer functions listed above for `FP16`.

1.5. On 3D hardware that supports `FP16` as vertex components, modify one of the basic drawing demos in this book to use `FP16` vertices. This will require some modifications and additions to the engine code handling of vertices. Use your functions above to convert the vertex data to `FP16`.

Chapter 2

2.1. For the following pairs of vectors, compute the following: their sum, their dot product, and their cross product:

1. $\mathbf{v} = (-2, 5, 10)$, $\mathbf{w} = (1, -2, 4)$
2. $\mathbf{v} = (6, -7, 4)$, $\mathbf{w} = (-6, 7, -4)$
3. $\mathbf{v} = (4, 8, -1)$, $\mathbf{w} = (-2, 2, 8)$
4. $\mathbf{v} = (1, 3, -2)$, $\mathbf{w} = (-4, -12, 8)$

2.2. Of the following sets of vectors, which could be used as a basis for $\mathbb{R}^3$? For those that could be used as a basis, perform Gram-Schmidt orthogonalization to get an orthonormal basis. Which are left handed and which are right handed?

1. $\mathbf{u} = (-6, 1, 6)$, $\mathbf{v} = (-5, -9, -2)$, $\mathbf{w} = (3, -6, 1)$
2. $\mathbf{u} = (4, -10, 4)$, $\mathbf{v} = (9, 1, 5)$, $\mathbf{w} = (2, -3, -3)$
3. $\mathbf{u} = (1, -2, -5)$, $\mathbf{v} = (-10, 3, 7)$, $\mathbf{w} = (-7, -3, -8)$
4. $\mathbf{u} = (0, 3, 0)$, $\mathbf{v} = (-9, 6, 10)$, $\mathbf{w} = (-8, 3, 5)$

2.3. Taking the arccosine of the dot product can give us an unsigned angle between two vectors; that is, the angle is always positive whether we are rotating from $\mathbf{v}$ to $\mathbf{w}$, or from $\mathbf{w}$ to $\mathbf{v}$. It's also possible to have a signed angle, where the rotation direction matters. In this case the angle from $\mathbf{v}$ to $\mathbf{w}$ is the negative of the angle from $\mathbf{w}$ to $\mathbf{v}$. Come with a method of computing a signed angle between two vectors. Hint: Consider the definition of the arctangent.

2.4. One way of thinking about the convexity of a set of points is to consider each point as a nail in a board. If we can place a single rubber band around the set of nails and the band touches all of the nails, then the set is convex, otherwise it is concave. Another way to represent this is to tie a piece of string to one nail and then start sweeping the string around each nail in turn, say in a clockwise direction. If we ever need to change direction from clockwise to counterclockwise to get the next nail, then the set is concave.

1. Based on this description, outline a routine to determine whether a set of 2D points is convex or concave. Hint: Use the perpendicular dot product as a measure of winding direction.
2. Modify this routine to compute the convex hull of the set of points.

2.5. For each set of points compute a parameterized line that contains them all (if possible):

1. $P = (10, -9, -4)$, $Q = (-5, 7, -9)$
2. $P = (-8, 5, 8)$, $Q = (-5, -9, 4)$
3. $P = (9, -10, 2)$, $Q = (2, 2, -7)$, $R = (-10, 1, 3)$
4. $P = (1, 6, 7)$, $Q = (3, -3, 6)$, $R = (-1, 8, 3)$

2.6. For each set of points compute a plane that contains them all (if possible). Use both the parameterized and general plane equations.

1. $P = (1, 10, 0)$, $Q = (9, 6, 3)$, $R = (10, 9, 1)$
2. $P = (-4, -2, 10)$, $Q = (1, 10, 0)$, $R = (2, 1, -10)$
3. $P = (-8, 6, 6)$, $Q = (9, -6, 3)$, $R = (-10, 4, 8)$, $S = (14, -8, 6)$
4. $P = (2, 0, 10)$, $Q = (-3, -1, -7)$, $R = (-5, -5, 17)$, $S = (-4, -3, 5)$

2.7. Determine algebraically which of the following points lie within the triangle represented by $P = (0, 5, -1)$, $Q = (4, 1, 2)$, $R = (-3, -1, 2)$:

1. $\mathbf{u} = (1, 1, 1.5)$
2. $\mathbf{u} = (3.25, 4, -0.75)$
3. $\mathbf{u} = (0, -1, 1)$
4. $\mathbf{u} = (0.5, 0, 2)$

Chapter 3

3.1. Multiply the following matrices together:

1. $\begin{bmatrix} 1 & -2 \\ 7 & 5 \end{bmatrix} \begin{bmatrix} -3 & 9 \\ -4 & 1 \end{bmatrix}$
2. $\begin{bmatrix} -10 & 2 \\ -5 & -9 \end{bmatrix} \begin{bmatrix} 3 & 6 \\ -1 & -4 \end{bmatrix}$
3. $\begin{bmatrix} -5 & -9 & -5 \\ -6 & 5 & 4 \\ 1 & 6 & 7 \end{bmatrix} \begin{bmatrix} -6 \\ -3 \\ 8 \end{bmatrix}$
4. $\begin{bmatrix} 1 & -2 & -5 \\ -10 & 3 & 7 \\ -7 & -3 & -8 \end{bmatrix} \begin{bmatrix} -6 & 5 & 4 \\ 3 & -3 & 6 \\ -8 & 5 & 8 \end{bmatrix}$

3.2. In Section 3.2.5, we state that $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$. Expand this for a general matrix and prove that this is true.

3.3. In Section 3.2.7, we introduce the tensor matrix and the formula $(\mathbf{u} \cdot \mathbf{v})\mathbf{w} = (\mathbf{w} \otimes \mathbf{v})\mathbf{u}$. Expand the terms and prove that this is true.

3.4. Using the formula $\mathcal{T}(a\mathbf{x} + \mathbf{y}) = a\mathcal{T}(\mathbf{x}) + \mathcal{T}(\mathbf{y})$, determine whether the following are linear transformations:

1. $\mathcal{T}(\mathbf{x}, \mathbf{y}) = 2x + y$
2. $\mathcal{T}(\mathbf{x}, \mathbf{y}) = 2x + 1$
3. $\mathcal{T}(\mathbf{x}, \mathbf{y}) = (2x, 1)$
4. $\mathcal{T}(\mathbf{x}, \mathbf{y}) = (2x^2 + y, y)$
5. $\mathcal{T}(\mathbf{x}, \mathbf{y}) = (\cos\theta x + \sin\theta y, -\sin\theta x + \cos\theta y)$

3.5. Solve the following linear systems (if possible):

1. $$\begin{array}{rcrcr} x & + & 2y & = & 1 \\ 2x & + & y & = & -1 \end{array}$$
2. $$\begin{array}{rcrcr} 3x & + & 5y & = & -3 \\ x & + & 2y & = & 4 \end{array}$$
3. $$\begin{array}{rcrcr} 3x & - & y & = & 5 \\ x & + & y & = & 2 \end{array}$$
4. $$\begin{array}{rcrcrcr} x & + & 6y & + & 7z & = & 1 \\ 3x & - & 3y & + & 6z & = & -1 \\ -x & + & 8y & + & 3z & = & 2 \end{array}$$
5. $$\begin{array}{rcrcrcr} 2x & + & 8y & + & 3z & = & 1 \\ & - & 8y & + & 3z & = & 3 \\ & - & 4y & + & 3z & = & 3 \end{array}$$

3.6. Invert the following matrices (if possible):

1. $$\begin{bmatrix} 1 & -2 \\ 7 & 5 \end{bmatrix}$$
2. $$\begin{bmatrix} -10 & 2 \\ -5 & -9 \end{bmatrix}$$
3. $$\begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{bmatrix}$$

4. $\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{bmatrix}$
5. $\begin{bmatrix} 1 & -2 & -5 \\ -10 & 3 & 7 \\ -7 & -3 & -8 \end{bmatrix}$

3.7. Compute the determinant, eigenvectors, and eigenvalues for the following 2×2 matrices:

1. $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$
2. $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$
3. $\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$
4. $\begin{bmatrix} 2 & -1 \\ 1 & -2 \end{bmatrix}$

Chapter 4

4.1. Build a matrix that performs the following transformations in their given order. Assume that we're using column vectors.

1. Rotation around the x-axis by 45 degrees, then translation by $(-6, 7, -4)$.
2. Rotation around the y-axis by -45 degrees, then z shear by $(-6, 7, 0)$.
3. Scale in x by 2, translation by $\mathbf{v} = (-6, 7, -4)$, then rotation around x by 90 degrees.
4. Reflection across the xy plane, rotation around z by 90 degrees, and then an x shear by $(0, -4, 2)$.

4.2. In Section 4.3.2 we derive one form of a generalized rotation matrix. Generate a similar matrix for rotation order $\mathbf{R}_z\mathbf{R}_x\mathbf{R}_y$.

4.3. Derive the general matrix for plane reflection at an arbitrary point. Show that this is the same as translating that point to the origin, performing the reflection, and translating back to the center of transformation.

Chapter 5

5.1. Compute *z*-*y*-*x* Euler angles for the following rotation matrices:

1. $$\begin{bmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2. $$\begin{bmatrix} \frac{1}{2} & -\frac{3}{4} & -\frac{\sqrt{3}}{4} \\ \frac{\sqrt{3}}{2} & \frac{\sqrt{3}}{4} & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix}$$

3. $$\begin{bmatrix} 0.7426 & -0.3038 & -5.969 \\ 0.02887 & 0.9049 & -0.4247 \\ 0.6691 & 0.2982 & 0.6807 \end{bmatrix}$$

Verify that a vector rotated by the given matrices and their associated Euler angles ends up the same.

5.2. Compute rotation axis and angle for the matrices in question 5.1. Verify that a vector rotated by the given matrices and their associated rotation axis and angle ends up the same.

5.3. Compute equivalent quaternions for the matrices in question 5.1. Verify that a vector rotated by the given matrices and their associated quaternions ends up the same.

5.4. Given the following rotation axes and angles (in degrees), compute the corresponding quaternion:

1. $\mathbf{r} = (-1, 0, 0), \theta = 90$
2. $\mathbf{r} = (-1, 1, 0), \theta = 45$
3. $\mathbf{r} = (-1, 2, 1), \theta = 30$
4. $\mathbf{r} = (-3, 0, 1), \theta = 60$

Verify that a vector rotated by the given rotation angles and axes and their associated quaternions ends up the same.

5.5. Given the following, compute the quaternion that will rotate vector $\mathbf{u}$ into vector $\mathbf{v}$:

1. $\mathbf{u} = (-1, 0, 0)$, $\mathbf{v} = (0, 1, 0)$
2. $\mathbf{u} = (-1, 0, 0)$, $\mathbf{v} = (1, 0, 0)$

3. $\mathbf{u} = (0, -1, 1)$, $\mathbf{v} = (0.5, 0, 2)$
4. $\mathbf{u} = (1, -2, 4)$, $\mathbf{v} = (1, 3, -2)$

Chapter 6

6.1. In this chapter, we derived the standard OpenGL viewing matrix, perspective projection matrix, orthographic projection matrix, and oblique projection matrix, and merely presented the D3D matrices. Given what you know about the D3D standard viewing and NDC spaces, derive the corresponding matrices for the standard D3D pipeline.

6.2. Mathematically, there is no reason why the view space basis needs to have the x-axis representing "up," the y-axis representing "side," and the z-axis either "forward" or "back" depending on whether we're using the standard or OpenGL viewing representation, respectively. An alternative right-handed viewing system has y as up, z as right, and x as forward. Derive OpenGL-compatible viewing and perspective projection matrices using this system. Assume non oblique projection.

6.3. The discussion in the book assumes that we wish to maintain compatibility with the standard D3D or OpenGL fixed-function matrices. However, as question 6.2 shows, that is not necessary. In fact, it is almost possible to use the standard D3D matrices with OpenGL and vice versa. Why, and in what, cases, does this fail? (Playing with the projection matrix example code might make this easier to think about.)

6.4. In our stereo example, we use two oblique perspective projections to represent the viewpoints of the left and right eyes. These projections are set to use the same view plane and view window at a fixed distance from the viewer, but with different eyepoints. Now suppose that we want to have our game track an object of interest, and treat its location as the focal point of our stereo projection. To do this, we'll want to put the view plane so it passes through the position of the object. Modify the stereo example, using the teapot as the focus of our attention.

Implemented blindly, this scheme can fail in certain circumstances. What are those circumstances, and how can they be dealt with?

6.5. Another problem with stereo is placing a user interface. If we use a standard orthographic projection while overlaying the interface over the scene, there will be a disconnect between the visual cues of occlusion and depth perception. Occlusion tells our brain the UI is on top, but by drawing it as a single element our brain believes that the UI is on the projection plane, which may be further into the screen. The solution is to draw the UI at the near plane. There are two possible approaches to this: Either determine the necessary separation at the near plane and render with an orthographic projection twice, translating the UI appropriately, or render the UI using the perspective projection and scaling it to fit the screen. Choose one approach and rewrite the stereo example to implement this. Note: Since the stereo example uses the red-blue color scheme, your interface will have to be in grayscale.

6.6. In the clipping example, the `IvClipper` class only clips against a single plane. However, most APIs support clipping against multiple user planes. As mentioned, this process involves clipping the object against successive planes until the final result is achieved. One possibility is to generate the entire clipped polygon and then pass it through the clipper with a different plane equation, but this is not necessary. It also can use an indeterminate amount of memory, which is not ideal for a hardware graphics pipeline. Instead it is possible to pipeline the process, so the output of one clipper is passed as the input for another. Using this knowledge, rewrite the Clipper example to clip against multiple planes.

Chapter 7

7.1. As discussed in the text, indexed geometry can promote vertex reuse and can minimize vertex buffer size while improving performance. However, in the real world, many datasets are delivered in nonindexed "three vertex per triangle" format, even though many of those vertices are duplicated. Consequently, programmers creating real-time 3D tool and rendering pipelines are left with the need to generate optimal indexed geometry from nonindexed geometry. This exercise is one of creating such a set of functions.

1. First, we need to generate a suboptimal dataset. Modify one of the indexed rendering examples in the book's demos (specifically, one of the cylinder-rendering texture demos) to use the index array to expand the vertex array to $numTriangles \times 3$ vertices, and render the result without an index list. The result should look the same as before, but will use more vertices.
2. Using this nonoptimal vertex array as the source, add code into the demo to optimize the "bad" data. Specifically, create a new (presumably shorter) array of vertices and an index array that, when used together to draw an indexed triangle list, will render exactly the same object. This involves determining the set of truly unique vertices

and a remapping of the vertices such that the same triangles are drawn. Thus, the resulting index list will have the same length as the original index array that the demo used before we "unoptimized" the data. The resulting data may have fewer vertices than the original demo (before we duplicated the vertices in the first step of the exercise). Why would that be?

3. If you know that dynamic lighting is not to be used (and thus the normals are not to be referenced), how can you modify the optimization code to render correctly and merge even more vertices?

7.2. Triangle strips have a fixed topology: the "strip" shape. As a result, it is rare that a general geometry object can be rendered as a single, optimal strip. As discussed in the text, a common method is to generate geometry as a set of multiple strips, stopping each strip when there are no new triangles that can be added in strip layout. This can lead to numerous draw calls per object, and lowers the value of stripped data. Another trick that can be used is to "stitch" multiple indexed triangle strips together using the following observation: Modern rendering APIs normally guarantee that any triangle in a strip that does not have three unique indices will render no pixels. In other words, the triangle 1, 2, 2, in a strip will render nothing. Given this fact, show a method to stitch a strip that ends with the indices A, B, C to a strip that starts 1, 2, 3. Avoid adding more indices than needed, avoid rendering any pixels not rendered by the original set of two strips, and avoid any clockwise/counterclockwise issues. To solve the lattermost issue, what more information do you need to know about the strip(s).

7.3. In the text, we describe the method for computing the luminance of an RGB color. By replicating this luminance result across R, G, and B components, we can create a grayscale image.

1. Convert one of the basic, textured demo shaders to output the luminance grayscale instead of the color result.
2. Extend this shader with a "fader" uniform and use this [0, 1] range uniform to cross-fade smoothly from RGB to grayscale. Hook the uniform value to a set of keys and show the fading.

7.4. "Palettized" or "Paletted" textures are a form of texturing that was very popular in software renderers and early hardware renderers. They involve two components: and image and a "palette." The image is a 2D image, like a modern texture, but very compact, normally 8 bits (256 values) per texel. We call the texel values "indices." The palette is a lookup table with as many entries as the texture has values (thus, 256 in most cases). Each of the entries in the lookup table is a full color, such as a 32-bit RGBA value. To texture from a palettized texture, one finds the texel value at the desired location and then looks up the palette value of the resulting index, giving the color. Note that the palette could be shared between multiple textures, saving even more memory. This exercise discusses some of the issues of palettized textures.

1. Compare the memory sizes of an $M \times N$ 32-bit-per-texel texture with an $M \times N$ 8-bit palettized texture, including the size of the 256-entry, 32-bit palette. Write the formula for the size of each texture.
2. Describe how to compute bilinear filtering with a palettized texture.
3. Some modern 3D hardware does not support palettized textures directly. Write a fragment shader that implements a palettized texture as a pair of nonpalettized textures. Hint: Think indirect texturing.

Chapter 8

8.1. We have shown several situations where textures can be used to help simulate lighting by replacing material values, representing surface orientation (normal mapping), etc. However, textures also can be used to simulate lighting-related functions. With per-fragment specular lighting, we must compute the dot product of the view and reflection vectors, and then clamp the result to zero and apply the power function in order to simulate specular falloff. The power function can be expensive on some hardware, and is still somewhat limiting. The result of the dot product in question will always be in the range $[-1, 1]$, and that we clamp values below 0 to 0. These facts make it quite easy and convenient to use texturing (whose UV values are *not* constant and attached to the geometry) to replace the power function.

1. Create a fragment shader that computes per-fragment specular lighting without calling the power function. Use a $1 \times N$–texel texture to represent the power function.
2. How could you expand this method to allow a uniform to specify a different specular power? Hint: Use a 2D texture.

8.2. Fogging, or "depth cuing," is a popular effect that involves fading between the desired shaded object color and a constant color (the fog color) based on some notion of distance from the viewer. Fogging is most frequently used with the background or clear color as the fogging color, so that distant objects recede into the fog and can then be culled away. This is particularly useful when rendering a large, expansive environment, as it allows all objects beyond the maximum fog distance to be culled. It can be computed in multiple ways.

1. Modify the vertex shader of one of the more advanced shading demos to compute fog based on the *Z* value in camera space. Assume that the input uniforms are minimum depth for fogging (the depth at which fogging starts) and the maximum fog depth (where the fog value should reach 1). Pass the fog value, which should be in the range $[0, 1]$, to the fragment shader as a varying. In the fragment shader, use the fog varying value and another uniform, the fog color, to implement the actual fogging. Blend between the originally assigned fragment color and the fog color, based on the varying fog value. Adjust the fog color and minimum and maximum depths and move the scene objects closer and further from the viewer.
2. Depth or *Z* fog is based entirely on the *Z* value in camera space. This can lead to artifacts as the viewer turns the camera from side to side while standing in place. Visible objects toward the left and right edges of the view can actually disappear into the fog when the camera turns to look directly at them. This is because *Z* is not equal to the distance from the camera location to the object; it is only the distance from the view *plane* to the object, and thus changes as the view plane rotates. Thus, another form of fogging is called "range fog," and it is based on the distance (or the squared distance) from the camera center to the object-space vertex position. Modify your vertex shader above to implement range fog instead of depth fog. If possible, connect a key in the demo to switch between the two modes to see the difference.

8.3. As discussed in the text, the given, unweighted method for generating surface normals from indexed geometry can lead to normals that are not representative of the smooth surface that is being approximated. The text mentions some methods for improving this system.

1. Write C code based on the book's pseudocode to generate normals from indexed geometry. Expand this code to compute, store, and use triangle-area weighting to bias the averaged normals.
2. There are other methods that can further improve on the area weighting. For example, triangle area may overrepresent thin triangles that happen to be long. Modify the above code to use weighting based on the angle between the two triangle edges at the vertex of interest.

8.4. Textures can be used to implement even more advanced materials than shown in the text. Expand the example "diffuse and gloss map" shader from the text as follows. Each of these steps can augment the previous:

1. Add an RGB emissive map, which replaces any emissive lighting term.
2. Split the diffuse and specular maps into a pair of RGB maps, to allow for different specular and diffuse tinting.
3. Use the specular map's alpha channel to encode "shininess": the specular exponent. Note that since the texture values are $[0, 1]$, you will need to use some other method to expand these values to a more reasonable specular exponent range.
4. Merge the above fogging code to the shader, using the per-pixel depth value (the normal texture alpha) as the fogging depth.

8.5. Deferred lighting is an effect whereby an application uses the ability to render not to the main screen, but to a texture the size of the screen (or several such textures). The effect "defers" final lighting by writing lighting-source values per pixel, rather than final colors. Once all of the objects have been rendered to these textures, the textures are used in an additional pass to render to the screen. During this final pass, the lighting is computed. Thus, the source textures to this final pass represent such information as surface normal, material color(s), and depth. Note that this means for a fixed view and objects, changing the lighting only requires re-rendering the final pass. For a large, static scene and view, this method allows for high-performance lighting, because no matter what the complexity of the scene, the expense of lighting is constant. For this exercise, expand the previous exercise's four shaders to add an RGBA normal-and-depth map (the normal XYZ is encoded as biased in the RGB components of the texture, and the depth is encoded in the alpha component). This new map replaces the varying values above that represented the normal and the depth at each fragment. This actually can be tested using the normal map-generation tools that are available in numerous image editing packages, which can treat a grayscale image as a height map and generate RGB-encoded normal maps based on the height map. The alpha channel can be based on the original height map, thus encoding depth.

Chapter 9

9.1. Some 3D hardware can support advanced texture formats (such as floating-point-valued textures), but cannot filter textures of these formats automatically. Bilinear filtering can be done manually in a fragment shader. Implement bilinear filtering in a fragment shader, given a texture and a 2-vector uniform describing the width and height of the texture. Do not be concerned with wrapping versus clamping. Be sure to set the built-in filtering mode to "nearest" to ensure that the filtering is being done only by the shader. This will (of course) require multiple texture samples in the shader to simulate a single, filtered texture sample.

9.2. Some texturelike effects can be better implemented using a procedural shader. A classic example is the black-and-white checkerboard texture. Using the simplest shader you can develop (avoiding conditionals where possible), implement a fragment shader that maps texture coordinates (assuming wrapped textures) into an $M \times N$ checkerboard (where M and N are uniforms to the shader) per UV unit square. Assume magnification (i.e., do not worry about filtering the texture). Apply the shader to one of the textured-quad demos supplied with the text.

9.3. As described in the text, there are numerous methods for computing the mipmap level of a given fragment and texture coordinate. The various screen-space texel versus pixel deltas at a given pixel must be transformed into a single mipmap-level value. Modern shader APIs make these texture deltas available to the fragment shader. In D3D's HLSL, the functions are `ddx` and `ddy`, both of which are multivalued (since they must return the U and V deltas over X and Y). In OpenGL's GLSL (but *not* OpenGL ES's GLSL-ES), the functions are `dFdx` and `dFdy`. These are the deltas that must be converted to a mipmap level as described in the text. Furthermore, the modern shader APIs also allow for the desired mipmap level to be specified when sampling a texture! In HLSL and GLSL, these are `tex2Dlod` and `texture2DLod`, respectively. Note that the HLSL version requires D3D's shader model 3, and thus some 3D hardware may not be able to support it. Also, the text's example code compiles shader code for shader model 2, so the programmer will need to modify this in order to make the example work in D3D. This exercise is an experiment and exploration. Implement several of the texture delta to texture LOD selection methods discussed in the text, as well as some of your own, and plug them into the text's mipmapping demo. See how the different mipmapping functions affect the rendered result, especially as the textured quads are seen in perspective. To test the basics, start by making the shader pass down an LOD that is much higher (coarser) than needed, so that the textured quad is obviously blurry. This will indicate that the required "texture lookup with explicit LOD" is supported by the hardware, driver, and shader model.

Chapter 10

10.1. Given the Hermite curve represented by the following points and tangents:

$$P_0 = (1, -2, 5)$$
$$P_1 = (4, 3, -3)$$
$$\mathbf{P}'_0 = (-2, 3, -1)$$
$$\mathbf{P}'_1 = (-2, 0, 1)$$

evaluate the curve at $u = 0.25$, 0.5, and 0.75.

10.2. When discussing autogeneration of Hermite splines, we covered two types: the clamped spline, where the end tangents are assigned by the user, and the natural spline, where the end tangents have a zero-valued second derivative. However, there are two other common examples.

The first is known as the *cyclic* end condition. This assumes that the first and second derivatives at the endpoints are equal. Note that this doesn't necessarily mean that the positions of the two endpoints have to be equal. Neither does it mean that the resulting curve will be symmetric if they are equal (i.e., you can't guarantee an oval). You might use a curve of this type if you want to ensure that the animated object ends up moving in the same direction at the end of the curve as it does at the beginning.

The second is the *acyclic* end condition. This is similar to the cyclic end condition, except that the first and second derivatives are negatives of each other. If the positions of the two endpoints are equal, this can produce a shape like the head of a tennis racket. You might use a curve of this type if you want to ensure that the animated object ends up moving in the opposite direction at the end of the curve as it does at the beginning.

Given these constraints, derive the system of linear equations for the cyclic and acyclic end conditions.

10.3. In Chapter 10, we noted that the standard cubic Catmull-Rom spline does not interpolate to the two endpoints, and we derived a quadratic equation that would fill in the gap from the first point to the second point. Perform the corresponding derivation for the quadratic curve between the next-to-last and last points.

10.4. Similar to the Catmull-Rom spline, the cubic Kochanek-Bartels spline is only valid between P_1 and P_{n-1}. To solve this problem in the example code we used a cyclic spline, where the tangent at P_0 is dependent on P_n

and P_1, and the tangent at P_n is dependent on P_{n-1} and P_0. However, as with the Catmull-Rom spline, we can replace this with quadratic curves between P_0 and P_1, and P_{n-1} and P_n. Derive the equations for the corresponding Kochanek-Bartels quadratic curves.

10.5. As mentioned in the text, we can have quadratic Bézier curves as well as the standard cubic Bézier curves. Examine the general definition of a Bézier curve and create the general equation for a quadratic Bézier curve and its corresponding matrix form.

10.6. Given the formula for a quadratic Bézier curve derived in question 10.5 and the control points P_0, P_1, P_2, perform the following:

1. Directly evaluate the curve at $t = 0.25$, 0.5, and 0.75.
2. Using forward differencing, evaluate at the same values.
3. Using de Castlejau's method, evaluate at the same values.

10.7. When using curved surfaces, the surface is implicitly represented by a control mesh — a grid of points with either tangents at each point (for a Hermite surface) or additional control points (for a Bézier surface). To generate a point on the surface, we interpolate along one direction of the grid (known as the u direction) to generate new control points, and then along the orthogonal direction (known as the v direction) using the interpolated control points. For Hermite surfaces we also need to interpolate the tangents in the u direction so that they can be used to interpolate in the v direction. And for lighting purposes, we also need to compute a normal at each generated point, which is computed by taking the cross product of the tangents in the u and v directions. So for both cases, we will need to interpolate tangents as well as positions.

1. Derive forward differencing equations for computing the tangents of a cubic Hermite curve, and a cubic Bézier curve.
2. Suppose we're using subdivision with Bézier curves. How can we compute the tangent at an interpolated point in this case?

Chapter 11

11.1. One thing that commonly occurs in sports is a streak of occurrences. For example, in baseball, batters often get "hot" and have streaks of hits.

Suppose the chance that a batter gets a hit during a game is 0.357. What is the probability that said batter will hit successfully in 56 games? What if his game hit percentage was 0.421? What would the probability of a 56-game streak be then?

11.2. Consider the following results from rolling two dice 360 times and summing them. Compute the chi-square values and describe what this means about the random number generator used.

Die Value	2	3	4	5	6	7	8	9	10	11	12
Experiment 1	6	19	28	31	48	69	58	37	30	23	11
Experiment 2	14	27	37	51	52	61	41	23	28	19	7

11.3. Suppose we have an 8-bit word computer, and we have the choice of the following random number generators:

1. `x = 17 * x;`
2. `x = 17 * x % 251;`
3. `x = 17 * x % 255;`
4. `x = 17 * x % 253;`
5. `x = 21 * x + 5;`
6. `x[i] = x[i-4]*x[i-7];`
7. `x[i] = x[i-4]*x[i-7];`

8.
   ```
   x[i] = x[i-4]+x[i-7]+c;
   c = (x[i-4]+x[i-7]+c) / 256;
   ```

9.
   ```
   a=21*(k&0xff)+(a>>4);
   b=37*(j&0xff)+(b>>4);
   x = (a << 4) + b;
   ```

For each case, and given what you've learned from the chapter, identify the generator type, and discuss whether it provides good random numbers. You may make certain assumptions about how each generator is initialized. If these were your only choices, which one would you use and why?

11.4. In Chapter 11, we derived methods to uniformly distribute points on the surface of a sphere or on a disc. Using either the rejection sampling approach or the spherical coordinates approach, derive a method for distributing points on a cylinder. Add this method to the SphereDisc example.

Chapter 12

12.1. Suppose we have two planes $\mathbf{n}_0 \cdot \mathbf{x} + d_0 = 0$ and $\mathbf{n}_1 \cdot \mathbf{x} + d_1 = 0$. If these two planes intersect, the result is a line. Since the line is perpendicular to both planes, the line direction will be parallel to $\mathbf{n}_0 \times \mathbf{n}_1$. To find a point on the line, we note that we can consider the line direction as the normal to a new plane, and $\mathbf{n}_0$ and $\mathbf{n}_1$ as basis vectors for that plane. Then the line point P can be represented as $P = u\,\mathbf{n}_0 + v\,\mathbf{n}_1$. P must also lie on the original two planes, so we can take our parameterization for P and substitute it into the two plane equations to get two linear equations, with two unknowns:

$$\begin{aligned}\mathbf{n}_0 \cdot (u\,\mathbf{n}_0 + v\,\mathbf{n}_1) + d_0 = 0\\ \mathbf{n}_1 \cdot (u\,\mathbf{n}_0 + v\,\mathbf{n}_1) + d_1 = 0\end{aligned}$$

Solve this linear system, and then write a routine to find the intersection between two planes. Be sure to handle special cases.

12.2. Write a routine to find the intersection between two 2D lines. Be sure to handle special cases. Hint: Recall that to find the intersection between two planes we substituted a parameterized representation into the general plane equations.

12.3. Recall that when we found the closest point between a line segment and a point, we broke the possibilities into three regions. If the point projected beyond either endpoint, then that endpoint was the closest point. Otherwise the result was the projected point in the middle of the segment. However, another approach could have been to calculate the parameter of the projected point, and clamp it to lie between 0 and 1, and then calculate the nearest point using that.

Now consider a 2D axis-aligned bounding box. To project a point to one of the sides of the box, say the "min" side that is normal to the x-axis, we check the point's x value against the bounding box's min value — if the point's value is less, we set it to the box's min value. We can do a similar procedure to project to the "max" side of the box. Now suppose this point is on the right side of the box, but above it. If we take this projected point and apply the same operation in the y direction, we'll end up clamping it down to the upper-right corner.

Using this projection and clamping operation, write a routine to compute the nearest point for an AABB.

12.4. Computing the closest point for an OBB is very similar to computing the closest point for an AABB. The main problem is that our point is world space, and we'd like it to be in the box's local space so we can perform our projection operation. We can represent our point P in box space as

$$P = C + x\mathbf{r}_0 + y\mathbf{r}_1 + z\mathbf{r}_2$$

where C is the center of the box and the $\mathbf{r}_i$ vectors are the columns of the box's rotation matrix. We can rewrite this as

$$P - C = x\mathbf{r}_0 + y\mathbf{r}_1 + z\mathbf{r}_2$$

Using this equation, solve for the point's x, y, and z in box space and then use that to write a routine to compute the nearest point for an OBB. Hint: What happens if you take the dot product of $\mathbf{r}_0$ with that equation?

12.5. There is no one bounding object that will work as a collision approximation for all geometry. Some geometry is best approximated by a box, some by a sphere. Because of this, most collision engines support multiple bounding object types, and detect collisions between all types of bounding objects. Using the basic principles of capsule–capsule and sphere–sphere collision, derive a method for detecting collision between a sphere and a capsule.

12.6. Extending our collision engine further, use the results from questions 12.3 and 12.4 to create methods for detecting collisions between a sphere and an AABB, and a sphere and an OBB.

Chapter 13

13.1. Given a 1D force function $F(x) = -0.5x$, starting point $x = 1.0$, velocity $v = 5.0$, and time step $dt = 0.1$, compute the next two positions and velocities using:

1. Standard Euler.
2. Midpoint method.
3. Verlet.
4. Symplectic Euler.

How does this change if $F(x) = -100x$? How does this change if $dt = 10$?

13.2. We are now given a 2D force function $\mathbf{F}(\mathbf{x}) = -0.5\mathbf{x}$, starting point $\mathbf{x} = (0.0, 1.0)$, velocity $\mathbf{v} = (5.0, -2.0)$, and time step $dt = 0.1$. Assume now that the force is being applied to a circular object, displaced from the center of mass by the radius vector $\mathbf{r} = (1.0, 0.0)$. Compute the next two positions, velocities, orientations, and angular momenta using:

1. Standard Euler.
2. Midpoint method.
3. Verlet.
4. Symplectic Euler.

How does this change if $\mathbf{F}(\mathbf{x}) = -100\mathbf{x}$? How does this change if $dt = 10$? How is this different from the linear case in question 13.1.?

13.3. Using the collision-detection routines written in the exercises for Chapter 12, generalize the orientation example to replace capsules with OBBs and add spheres.