High-Performance Graphics Techniques and Games Graphics

**Prerequisites**

A solid understanding of the concepts of computer graphics, and a knowledge of the details of the OpenGL graphics API, at a level of detail that allows you to consider alternatives to standard API approaches to creating animated, interactive images.

**Definitions**

Computer games are about many things, including storytelling, creating characters, maintaining databases of game features, and many general programming techniques to deliver maximum speed of operation to the player. One of the critical performance areas — because it’s one of the most compute-intensive bottlenecks in presenting the game to the player — is the graphics that present the game to the user. This is a different question than we’ve been dealing with in computer graphics to this point. Up to here, we have been focusing on the quality of the images while maintaining as much performance as we can, but in this chapter we focus on the performance of the programs while maintaining as much quality as we can. And this change makes a major difference in the kind of processes and techniques we use.

In a sense, this is not a new issue in computer graphics. For over 20 years, the computer field has included the area of “real-time graphics.” This originally focused on areas such as flight simulators, real-time monitoring of safety-critical system processes, and real-time simulations, often using the highest-power computers available at the time. Some of the real-time graphics processes also were used in educational applications that are essentially simulations. But the demands that games place on graphics are both as extreme as these and are focused on personal computers with widely varying configurations, making it important to bring these real-time techniques into a graphics text.

**Techniques**

Fundamentally, high-performance computer graphics, especially as applied to games, takes advantage of a few simple principles:

- Use hardware acceleration when you can, but don’t assume that everyone has it and be ready to work around it when you need to
- Do some work to determine what you don’t need to display
  - Look for techniques that will support easy ways to cull materials from display
- Take advantage of capabilities of texture mapping
  - Create objects as texture maps instead of as fully-rendered objects
  - Use multitextures and textures with both low and high resolution
- Use any techniques available to support the fastest display techniques you can
  - Display lists
  - Level of detail
- Avoid unnecessary lighting calculations
  - When you draw any object, only enable lights near the object
  - Use fog
- Collision detection

We will discuss these and show you as many techniques as we can to support them in helping you create high-performance graphics.

There are also some unique problems in some gaming situations that are not found in most other areas of graphics. The main one we will discuss is collision detection, because this is an area that
requires some simple computation that we can streamline in ways that are similar to the techniques discussed above.

**Hardware avoidance**

The computer graphics pipeline includes a number of places where there are opportunities to put hardware into the process to get more performance, and graphics cards have usually been quick to take advantage of these. When you use OpenGL on a system with such a card, the graphics system will probably use the hardware features automatically. Paradoxical as it may seem, however, relying on this approach to speed may not be the best idea for high performance. Parts of your audience might not have the kind of acceleration you are programming for, for example, and even hardware has its cost. But more fundamentally, sometimes using standard techniques and relying on the hardware to get you speed will be slower than looking for alternative techniques that can avoid the processing the card accelerates.

As an example, consider the Z-buffer that is supported on almost every graphics card. When you use the Z-buffer to handle depth testing, you must carry out reading, comparing, and writing operations for each pixel you draw. If you have a fast graphics card, this is higher-speed reading, comparing, and writing, of course, but it is faster to avoid these operations than to optimize them. There are some techniques we will talk about below that allow you to do some modest computation to avoid entire polygons or, even better, to avoid making depth tests altogether.

**Designing out visible polygons**

As you lay out the overall design of your image, you can ensure that there is only limited visibility of the overall scene from any viewpoint. This is part of the reason why one sees many walls in games — we see large polygons with texture mapping for detail, and only a very few polygons are visible from any point. The sense of visual richness is maintained by moving quickly between places with different and clearly understood environments, so that when the player makes the transition from one place to another, they see a very different world, and even though the world is simple, the constant changing makes the game seem constantly fresh.

Other techniques involve pre-computing what objects will be visible to the player from what positions. As a very simple example, when a player moves out of one space into another, nothing in the space being vacated can be seen, so all the polygons in that space can be ignored. This kind of pre-computed design can involve maintaining lists of visible polygons from each point with each direction the player is facing, a classical tradeoff of space for speed.

**Culling polygons**

One of the traditional techniques for avoiding drawing is to design only objects that are convex polyhedra (or can be made from collections of convex polyhedra) and then to identify those polygons in the polyhedra that are back faces from the eye point. Remember that the normal vector for the polygon points in the direction of the front (or outside) face, so that the polygon will be a front face if the normal vector points toward the eye, and will be a back face if the normal vector points away from the eye. Front faces are potentially visible; back faces are never visible. In terms of the diagram in Figure 13.1, with the orientation of the vectors N and E as shown, a front face will have an acute angle between the normal and eye vectors, so \(N \cdot E\) will be positive, and a back face will have an obtuse angle, so \(N \cdot E\) will be negative. Thus a visibility test is simply the algebraic sign of the term \(N \cdot E\).
Another kind of culling can take place on the viewing volume. Here you can compare each vertex of your polyhedron or polygon with the bounding planes on your view volume; if all of the vertices lie outside of the viewing volume based on comparisons with the same bounding plane, then the polyhedron or polygon cannot be seen in the defined view and need not be drawn. This calculation should be done after the viewing transformation so the boundaries of the view volume are easy to use, but before the polygons are actually rendered. Recalling that the viewing volume is a rectangular pyramid with apex at the origin and expanding in the negative Z-direction, the actual comparison calculations are given by the following:

\[
\begin{align*}
  y &> T*Z/ZNEAR \text{ or } y < B*Z/ZNEAR \\
  x &> R*Z/ZNEAR \text{ or } x < L*Z/ZNEAR \\
  z &> ZNEAR \text{ or } z < ZFAR
\end{align*}
\]

where \(T\), \(B\), \(R\), and \(L\) are the top, bottom, right, and left coordinates of the near plane \(Z = ZNEAR\) as indicated by the layout in the diagram in Figure 13.2 below.

Avoiding depth comparisons

One of the classic computer graphics techniques is to order your objects by depth and draw them from back to front, mimicking the way light would progress from objects to your eye. This is called the \textit{painter's algorithm}, and it was most popular when the Z-buffer was beyond the scope of most graphics programming. This technique can be relatively simple if your model is static, had no interlocking polygons, and was intended to be seen from a single viewpoint, because these make it easy to figure out what “back” and “front” mean and which of any two polygons is in front of the other. This is not the usual design philosophy for interactive graphics, however, and particularly...
for games, because moving geometry and moving eye points are constantly changing which things are in front of what others. So if we were to use this approach, we would find ourselves having to calculate distances from a moving eye point in varying directions, which would be very costly to do.

It may be possible to define your scene in ways that can ensure that you will only view it from points where the depth is known, or you may need to define more complex kinds of computation to give you that capability. A relatively common approach to this problem is given by binary space partitioning, as described below.

**Front-to-back drawing**

Sometimes a good idea is also a good idea when it is thought of backwards. As an alternative to the painter’s algorithm approach, sometimes you can arrange to draw objects only from the front to the back. This still requires a test, but you need test only whether a pixel has been written before you write it for a new polygon. When you are working with polygons that have expensive calculations per pixel, such as complex texture maps, you want to avoid calculating a pixel only to find it overwritten later, so by drawing from the front to back you can calculate only those pixels you will actually draw. You can use BSP tree techniques as discussed below to select the nearest objects, rather than the farthest, to draw first, or you can use pre-designed scenes or other approaches to know what objects are nearest.

**Binary space partitioning**

There are other approach to avoiding depth comparisons. It is possible to use techniques such as binary space partitioning to determine what is visible, or to determine the order of the objects as seen from the eyepoint. Here we design the scene in a way that can be subdivided into convex sub-regions by planes through the scene space and we can easily compute which of the sub-

![Figure 13.3: a collection of objects in a subdivided space](image-url)
regions is nearer and which is farther. This subdivision can be recursive: find a plane that does not intersect any of the objects in the scene and for which half the objects are in one half-space relative to the plane and the other half are in the other half-space, and regard each of these half-spaces as a separate scene to subdivide each recursively. The planes are usually kept as simple as possible by techniques such as choosing the planes to be parallel to the coordinate planes in your space, but if your modeling will not permit this, you can use any plane at all. This technique will fail, however, if you cannot place a plane between two objects, and in this case more complex modeling may be needed. This kind of subdivision is illustrated in Figure 13.3 for the simpler 2D case that is easier to see.

This partitioning allows us to view the space of the image in terms of a binary space partitioning tree (or BSP tree) that has the division planes as the interior nodes and the actual drawn objects as its leaves. With each interior note you can store the equation of the plane that divides the space, and with each branch of the tree you can store a sign that says whether that side is positive or negative when its coordinates are put into the plane equation. These support the computation of which side is nearer the eye, as noted below. This tree is shown in Figure 13.4, with each interior node indicated by the letters of the objects at that point in the space. With any eye point, you can determine which parts of the space are in front of which other parts by making one test for each interior node, and re-adjusting the tree so that (for example) the farther part is on the left-hand branch and the nearer part is on the right-hand branch. This convention is used for the tree in the figure with the eye point being to the lower right and outside the space. The actual drawing then can be done by traversing the tree left-to-right and drawing the objects as you come to them.

The actual test for which part is nearer can be done by considering the relation of the eye point to the plane that divides the space. If you put the eye coordinates into the plane equation, you will get either a positive or negative value, and objects on the side of the plane nearer the eye will have the same relation to the plane as the eye. Further, as your eye moves, you will only need to recompute the orientation of the BSP tree when your eye point crosses one of the partitioning planes, and you may be able to conclude that some of the orientations do not need to be recomputed at all.

If you have any moving objects in your scene, you must determine their relation to the other objects and account for them in relation to the BSP tree. It is common to have moving objects only show up in front of other things, and if this is the case then you can draw the scene with the BSP tree and simply draw the moving object last. However, if the moving object is placed among the other drawn objects, you can add it into the BSP tree in particular spaces as it moves, with much the same computation of its location as you did to determine the eye location, and with the object moved from one region to another when it crosses one of the dividing planes. Details of this operation are left to the reader at this time.
Clever use of textures

We have already seen that textures can make simple scenes seem complex and can give an audience a sense of seeing realistic objects. When we take advantage of some of the capabilities of texture mapping we can also deal with graphic operations in precisely the sense that we started this chapter with: reducing the accuracy in hard-to-see ways while increasing the efficiency of the graphics.

One technique is called billboarding, and involves creating texture-mapped versions of complex objects that will only be seen at a distance. By taking a snapshot — either a photograph or a once-computed image — and using the alpha channel in the texture map to make all the region outside the object we want to present blend to invisible, we can put the texture onto a single rectangle that is oriented towards the eye point and get the effect of a tree, or a building, or a vehicle, on each rectangle. If we repeat this process many times we can build forests, cities, or parking lots without doing any of the complex computation needed to actually compute the complex object. Orienting each billboard to eye point involves computing the positions of the billboard and the eye (which can be readily done from the scene graph by looking for translations that affect both) and computing the cylindrical or spherical coordinates of the eye point if the billboard is regarded as the origin. The latitude and longitude of the eye point from the billboard will tell you how to rotate the billboard so it faces toward the eye. Note that there are two ways to view a billboard; if it represents an object with a fixed base (tree, building, ...) then you only want to rotate it around its fixed axis; if it represents an object with no fixed point (snowflake) then you probably want to rotate it around two axes so it faces the eye directly.

Another technique is to use techniques at several levels of resolution. OpenGL provides a capacity to do mipmaps, texture maps at many resolutions. If you start with the highest-resolution (and hence largest) texture map, you can automatically create texture maps with lower resolution. Recall that each dimension of any texture map must be a power of two, so you can create maps with dimensions half the original, one fourth the original, and so on, yielding a sequence of texture maps that you can use to achieve your textures without the aliasing you would get if you used the larger texture.

Yet another approach is to layer textures to achieve your desired effects. This capability, called multitexturing, is only available for OpenGL at level 1.2 and beyond. It allows you to apply multiple textures to a polygon in any order you want, so you can create a brick wall as a color texture map, for example, and then apply a luminance texture map to make certain parts brighter, simulating the effect of light through a window or the brightness of a torch without doing any lighting computations whatsoever.

These last two techniques are fairly advanced and the interested student is referred to the manuals for more details.

System speedups

One kind of speedup available from the OpenGL system is the display list. As we noted in Chapter 3, you can assemble a rich collection of graphics operations into a display list that executes much more quickly than the original operations. This is because the computations are done at the time the display list is created, and only the final results are sent to the final output stage of the display. If you pre-organize chunks of your image into display lists, you can execute the lists and gain time. Because you cannot change the geometry once you have entered it into the display list, however, you cannot include things like polygon culling or changed display order in such a list.

Another speedup is provided by the “geometry compression” of triangle strips, triangle fans, and quad strips. If you can ensure that you can draw your geometry using these compression
techniques, even after you have done the culling and thresholding and have worked out the sequence you want to use for your polygons, these provide significant performance increases.

**LOD**

Level of Detail (usually just LOD) involves creating multiple versions of a graphical element and displaying a particular one of them based on the distance the element is from the viewer. This allows you to create very detailed models that will be seen when the element is near the viewer, but more simple models that will be seen when the element is far from the viewer. This saves rendering time and allows you to control the way things will be seen — or even whether the element will be seen at all.

Level of detail is not supported directly by OpenGL, so there are few definitions to be given for it. However, it is becoming an important issue in graphics systems because more and more complex models and environments are being created and it is more and more important to display them in real time. Even with faster and faster computer systems, these two goals are at odds and techniques must be found to display scenes as efficiently as possible.

The key concept here seems to be that the image of the object you're dealing with should have the same appearance at any distance. This would mean that the farther something is, the fewer details you need to provide or the coarser the approximation you can use. Certainly one key consideration is that one would not want to display any graphical element that is smaller than one pixel, or perhaps smaller than a few pixels. Making the decision on what to suppress at large distance, or what to enhance at close distance, is probably still a heuristic process, but there is research work on coarsening meshes automatically that could eventually make this better.

LOD is a bit more difficult to illustrate than fog, because it requires us to provide multiple models of the elements we are displaying. The standard technique for this is to identify the point in your graphical element (ObjX, ObjY, ObjZ) that you want to use to determine the element’s distance from the eye. OpenGL will let you determine the distance of any object from the eye, and you can determine the distance through code similar to that below in the function that displayed the element:

```c
glRasterPos3f( ObjX, ObjY, ObjZ );
glGetFloatv( GL_CURRENT_RASTER_DISTANCE, &dist );
if (farDist(dist)) { ... // farther element definition
} else { ... // nearer element definition
}
```

This allows you to display one version of the element if it is far from your viewpoint (determined by the a function float farDist(float) that you can define), and other versions as desired as the element moves nearer to your viewpoint. You may have more than two versions of your element, and you may use the distance that `glGetFloatv(GL_CURRENT_RASTER_DISTANCE, &dist)` returns in any way you wish to modify your modeling statements for the element.

To illustrate the general LOD concept, let's display a GLU sphere with different resolutions at different distances. Recall from the early modeling discussion that the GLU sphere is defined by the function

```c
void gluSphere (GLUquadricObj *qobj, GLdouble radius,
GLint slices, GLint stacks);
```

as a sphere centered at the origin with the radius specified. The two integers `slices` and `stacks` determine the granularity of the object; small values of `slices` and `stacks` will create a coarse sphere.
and large values will create a smoother sphere, but small values create a sphere with fewer polygons that’s faster to render. The LOD approach to a problem such as this is to define the distances at which you want to resolution to change, and to determine the number of slices and stacks that you want to display at each of these distances. Ideally you will analyze the number of pixels you want to see in each polygon in the sphere and will choose the number of slices and stacks that provides that number.

Our modeling approach is to create a function `mySphere` whose parameters are the center and radius of the desired sphere. In the function the depth of the sphere is determined by identifying the position of the center of the sphere and asking how far this position is from the eye, and using simple logic to define the values of `slices` and `stacks` that are passed to the `gluSphere` function in order to select a relatively constant granularity for these values. The essential code is

```c
myQuad=gluNewQuadric();
glRasterPos3fv( origin );
// howFar = distance from eye to center of sphere
glGetFloatv( GL_CURRENT_RASTER_DISTANCE, &howFar );
resolution = (GLint) (200.0/howFar);
slices = stacks = resolution;
gluSphere( myQuad , radius , slices , stacks  );
```

This example is fully worked out in the source code `multiSphere.c` included with this module. Some levels of the sphere are shown in Figure 13.5 below.

![Figure 13.5: levels of detail in the sphere, from high detail level at left to lower at right](image)

Reducing lighting computation

While we may include eight (or more) lights in a scene, each light we add takes a toll on the time it takes to render the scene. Recalling the lighting computations, you will recall that we calculate the ambient, diffuse, and specular lighting for each light and add them together to compute the light for any polygon or vertex. However, if you are using positional lights with attenuation, the amount of light a particular light adds to a vertex is pretty small when that vertex is not near the light. You may choose to simplify the light computation by disabling lights when they are not near the polygon you are working on. Again, the principle is to spend a little time on computation when it can offer the possibility of saving more time on the graphics calculation.

Fog

Fog is a technique which offers some possibility of using simpler models in a scene while hiding some of the details by reducing the visibility of the models. The tradeoff may or may not be worth doing, because the simpler models may not save as much time as it takes to calculate the effect of the fog. We include it here more because of its conceptual similarity to level-of-detail questions than for pure efficiency reasons.
When you use fog, the color of the display is modified by blending it with the fog color as the display is finally rendered from the OpenGL color buffer. Details of the blending are controlled by the contents of the depth buffer. You may specify the distance at which this blending starts, the distance at which no more blending occurs and the color is always the fog color, and the way the fog color is increased through the region between these two distances. Thus elements closer than the near distance are seen with no change, elements between the two distances are seen with a color that fades towards the fog color as the distance increases, and elements farther than the far distance are only seen with the full effect of the fog as determined by the fog density. This provides a method of depth cueing that can be very useful in some circumstances.

There are a small number of fundamental concepts needed to manage fog in OpenGL. They are all supplied through the `glFog*(param, value)` functions as follows, similarly to other system parameter settings, with all the capitalized terms being the specific values used for `param`. In this discussion we assume that color is specified in terms of RGB or RGBA; indexed color is noted briefly below.

**start and end:**
Fog is applied between the starting value `GL_FOG_START` and the ending value `GL_FOG_END`, with no fog applied before the starting value and no changes made in the fog after the end value. Note that these values are applied with the usual convention that the center of view is at the origin and the viewpoint is at a negative distance from the origin. The usual convention is to have fog start at 0 and end at 1.

**mode:**
OpenGL provides three built-in fog modes: linear, exponential, or exponential-squared. These affect the blending of element and fog color by computing the fog factor `ff` as follows:
- **GL_LINEAR:** \( ff = \text{density} \times z' \) for \( z' = (end-z)/(end-start) \) and any \( z \) between `start` and `end`.
- **GL_EXP:** \( ff = \exp(-\text{density} \times z') \) for \( z' \) as above
- **GL_EXP2:** \( ff = \exp(-\text{density} \times z')^2 \) for \( z' \) as above

The fog factor is then clamped to the range \([0,1]\) after it is computed. For all three modes, once the fog factor \( ff \) is computed, the final displayed color \( C_d \) is interpolated by the factor of \( ff \) between the element color \( C_e \) and the fog color \( C_f \) by \( C_d = ff \times C_e + (1-ff) \times C_f \).

**density:**
Density may be thought of as determining the maximum attenuation of the color of a graphical element by the fog, though the way that maximum is reached will depend on which fog mode is in place. The larger the density, the more quickly things will fade out in the fog and thus the more opaque the fog will seem. Density must be between 0 and 1.

**color:**
While we may think of fog as gray, this is not necessary — fog may take on any color at all. This color may be defined as a four-element vector or as four individual parameters, and the elements or parameters may be integers or floats, and there are variations on the `glFog*()` function for each. The details of the individual versions of `glFog*()` are very similar to `glColor*()` and `glMaterial*()` and we refer you to the manuals for the details. Because fog is applied to graphics elements but not the background, it is a very good idea to make the fog and background colors be the same.

There are two additional options that we will skim over lightly, but that should at least be mentioned in passing. First, it is possible to use fog when you are using indexed color in place of RGB or RGBA color; in that case the color indices are interpolated instead of the color.
specification. (We did not cover indexed color when we talked about color models, but some older graphics systems only used this color technology and you might want to review that in your text or reference sources.) Second, fog is hintable — you may use `glHint(...)` with parameter `GL_FOG_HINT` and any of the hint levels to speed up rendering of the image with fog.

Fog is an easy process to illustrate. All of fog’s effects can be defined in the initialization function, where the fog mode, color, density, and starting and ending points are defined. The actual imaging effect happens when the image is rendered, when the color of graphical elements are determined by blending the color of the element with the color of fog as determined by the fog mode. The various fog-related functions are shown in the code fragment below.

```c
void myinit(void)
{
    static GLfloat fogColor[4]={0.5,0.5,0.5,1.0}; // 50% gray
    ...
    // define the fog parameters
    ...
    glFogi(GL_FOGMODE, GL_EXP); // exponential fog increase
    glFogfv(GL_FOGCOLOR, fogColor); // set the fog color
    glFogf(GL_FOGSTART, 0.0); // standard start
    glFogf(GL_FOGEND, 1.0); // standard end
    glFogf(GL_FOGDENSITY, 0.50); // how dense is the fog?
    ...
    glEnable(GL_FOG); // enable the fog
    ...
}
```

An example illustrates our perennial cube in a foggy space, shown in Figure 13.6. This builds on the earlier textured cube to include fog in addition to the texture map on one face of the cube. (The texture map itself is included with this module; it is a screen capture of a graphic display, saved in Photoshop™ as a raw RGB file with no structure.) The student is encouraged to experiment with the fog mode, color, density, and starting and ending values to examine the effect of these parameters’ changes on your images. This example has three different kinds of sides (red, yellow, and texture-mapped) and a fog density of only 0.15, and has a distinctly non-foggy background for effect.

![Figure 13.6: a foggy cube (including a texture map on one surface)](image)

**Collision detection**

When you do polygon-based graphics, the question of collisions between objects reduces to the question of collisions between polygons. By reducing the general polygon to a triangle, that further reduces to the question of collisions between an edge and a triangle. We actually
introduced this issue earlier in the mathematical background, but it boils down to extending the edge to a complete line, intersecting the line with the plane of the polygon, and then noting that the edge meets the polygon if it meets a sequence of successively more focused criteria:

- the parameter of the line where it intersects the plane must lie between 0 and 1
- the point where the line intersects the plane must lie within the smallest circle containing the triangle
- the point where the line intersects the plane must lie within the body of the triangle.

This comparison process is illustrated in Figure 13.7 below.

If you detect a collision when you are working with moving polyhedra, the presence of an intersection might require more processing because you want to find the exact moment when the moving polyhedra met. In order to find this intersection time, you must do some computations in the time interval between the previous step (before the intersection) and the current step (when the intersection exists). You might want to apply a bisection process on the time, for example, to determine whether the intersection existed or not halfway between the previous and current step, continuing that process until you get a sufficiently good estimate of the actual time the objects met. Taking a different approach, you might want to do some analytical computation to calculate the intersection time given the positions and velocities of the objects at the previous and current times so you can re-compute the positions of the objects to reflect a bounce or other kind of interaction between them.

![Figure 13.7: the collision detection computation](image)

A word to the wise...

As LOD techniques are used in animated scenes, you must avoid sudden appearance or disappearance of objects as well as sudden jumps in appearance. These artifacts cause a break in the action that destroys the believability of the animation. It can be useful to create a fog zone deep in a scene and have things appear through the fog instead of simple jumping into place.

Fog is a tempting technique because it looks cool to have objects that aren't as sharp and “finished” looking as most objects seem to be in computer graphics. This is similar to the urge to use texture mapping to get objects that don’t seem to be made of smooth plastic, and the urge to use smooth-shaded objects so they don’t seem to be crudely faceted. In all these cases, though, using the extra techniques has a cost in extra rendering time and programming effort, and unless the technique is merited by the communication needed in the scene, it can detract from the real meaning of the graphics.