Lecture VI
PIXEL MANIPULATION

§1. Introduction

- There two fundamental approaches to visual representation: geometry and pixel maps. The former is powerful when used in the appropriate domaint. Thus, there is no better way to represent a perfect sphere than to use geometry. But there are times when we simply do not have enough information to specify the geometry or the necessary geometry is too complex. Thus, a geometric representation of a tree or mountain is generally not a good idea. Pixels (and more generally, some simple grid representation) is more useful for such representation.
- Geometric models are more brittle and harder to come by. But pixel models (images) are easily acquired and are by far much more abundant. Most physical devices acquire data in this format.
- Let us make a connection to an issue we know from homework 1. There you displayed geometric data (TIGER maps) on a window. When you implement zooming and panning, these displays did not work that well because you had to redraw the images each time. We will see that if we used pixel manipulation primitives, these effects can be achieved in a more satisfactory way.
- Topics we will address in this lecture includes: operations on pixmaps, compositing images, anti-aliasing, z-buffers, alpha-blending, etc.

§2. Fragments and Pixels

- In our previous consideration of the OpenGL graphics pipeline, we followed the transformations of the geometry from the model through to the viewport transformation.
- We now want to follow the pipeline beyond this stage. At this point, the graphics engine has enough information to begin the **rasterization** process (a.k.a. scan conversion).
- Rasterization will ultimately produce **pixels** that are values for specific positions on the screen. But in general, the result of rasterization are intended for an abstraction of the screen, called **buffers**). They can be hardware memory units that are closely related to the screen.
- We can think of pixels as the ultimate goal of rasterization, but the are intermediate objects are called **fragments**. We can regard a fragment as a (r, g, b, z, t, s) value where r, g, b are color components, z is the depth information and t, s are texture coordinates. Each fragment has a 2 dimensional coordinate (x, y).
- There is no simple connection between fragments and pixels. Several fragments can contribute to one pixel, and one fragment can contribute to several pixels.
- Another abstraction of the screen is 2 dimensional array of pixels, which we call a pixel map or **pixmap**. The pixels values are typically (r, g, b) or (r, g, b, a) values. A special case of pixmaps is a **bitmap** where each value is Boolean.

§3. Frame Buffers

- An OpenGL **buffer** is an array that is dimensioned as the screen, and stores a per-pixel set of data. There are several buffers in OpenGL: altogether, they comprise the **frame buffer**.
- The most obvious buffers are the **color buffers**, which are meant to be displayed. The four main color buffers are called **front-left**, **front-right**, **back-left** and **back-right**. At any moment, the front-left buffer (if monoscopic), or the front-left and front-right buffers (if stereoscopic), will be displayed.
- The remaining color buffers (depending on implementation) are called **auxilliary buffers** (numbered 1, 2, 3, etc). To query the number of auxiliary buffers, use `glGetIntegerv()` with `GL_AUX_BUFFERS`.
- The front-back designation for color buffers are meant for **double buffering**, a technique to ensure smooth display transitions in animations. See below. If double-buffering is not used, only the **front buffers** are active. The left-right designation are meant for stereoscopic viewing. If stereo viewing is not used, only the **left buffers** are active. To find out if double buffering and/or stereoscopic buffers are available, use `glGetBooleanv()` with `GL_STEREO` or `GL_DOUBLEBUFFER`.
- Clearing buffers – this is an expensive operation because of the size of buffers. Hence, special instructions are given for this. As a buffer is cleared, a pre-specified "clearing color" will be loaded into the buffer. For the color buffers, the clearing color is specified by invoking `glClearColor(r, g, b)`. To actually clear, we call `glClear()`.
- In addition, we have the **depth**, **stencil**, **accumulation** buffers. These are somewhat implementation dependent, including how many bits per pixel in buffer. But you can query the system for all such information.
- There are many uses for these buffers: principally, depth buffer is useful for hidden-surface removal, accumulation buffer is useful for antialiasing and compositing, stencil buffers are useful for stenciling.

§4. Color Buffers

- The normal display method is to display and write on the same color buffer. This method is called **single buffering**. The alternative is called **double buffering**, a technique essential for animation.
- The idea is that while the front buffer is being displayed, the graphics is drawing the next frame in the back buffer. When the drawing is done, the front and back buffers are swapped, thus updating the scene. This ensures a smooth transition (otherwise, the frame is changed the minute the drawing is done, and you see mostly partially drawn images).
- You choose to use double buffering combined with depth buffer by calling `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB) | DEPTH_DEPTH` as part of your window initialization. The default argument is `GLUT_SINGLE | GLUT_RGBA`. Other possible arguments which can be or'd into the mask are: `GLUT_STENCIL`, `GLUT_ACCUM`, `GLUT_INDEX`.
- Swapping buffers may not be available and is a hardware-dependent operation. In GLUT, this is called as `void glutSwapBuffers(void)`.
- What if you need to switch between single buffering and double buffering? The reason for wanting this may be because in double buffering you have fewer colors available than in single buffering. Perhaps double buffers only support 16 colors (4 bits) which single buffers have 256 colors (8 bits). But switching is not easy on most window systems. The solution is to use two windows, with single and double buffering on each. You then use some windowing method to make exactly one of them visible.

§5. Depth Buffers

- In general, each buffer has a "standard" application. But there will be many non-standard tasks that we can accomplish using the buffers (call them tricks or techniques, depending on your view!). What distinguishes one class of buffers from another are the kinds of tests and pixel-wise manipulations defined for them. The best ways to learn the properties of these buffers is to see them in action.

- The main purpose of the depth buffer is hidden surface elimination. Let us describe this process. We will also show a special trick for drawing decals.

- Each fragment that gets written on the color buffer has a z-value that represents distance from the eye. If the depth test is enabled, then the z-value of each fragment is first compared ot the z-value. If the z-value of the fragment is less than the current z-value at the fragment position, then the fragment is placed in the color buffer. All these default behaviour can be modified.

- Let us consider the following problem: suppose we want to display a decal (or texture) in front of a geometric model. If the decal is placed substantially in front of the model, this is easy. What if the decal is a pixmap that must be rendered on the face F of a polygon? That is, they are coplanar.

- Here is the standard hidden surface removal solution:
  (1) Enable the depth buffer.
  (2) Draw F.
  (3) Draw the decal.

  Unfortunately, in step (3), because of rounding errors, one cannot reliably predict whether P or the decal would be displayed at any pixel position where they both appear. So this does not work. Of course, the above solution will work if we move the decal slightly in front of F. But then, a side view of the model will appear odd.

- Here is a better solution:
  (1) Enable depth test
  (2) Disable writing to depth buffer and render F.
  (3) Enable writing to depth buffer and render decal.
  (4) Disable writing to color buffer and render F again.
  (5) Enable writing to color buffer.

- Why does this work? In step (2), F is properly rendered with respect to the rest of the scene. However, the depth buffer is not modified. In step (3), decal is also properly rendered with respect to the rest of the scene because its depth test is using the depth value of the original scene. Moreover, the decal's fragment will always override the corresponding fragment from F, as intended. Step (4) is performed in order to ensure that the depth buffer is now properly updated (which should have happened in step (2) had we not turned off depth buffer writing). Step (5) is to return to normal state of writing to color buffers.

- It is important to make explicit some assumptions in this solution:
  (i) Depth buffer comparison between F and decal is unreliable. But notice that in our solution above, we never compare the depth value from F with the depth value from decal.
  (ii) Depth buffer comparison between F or decal with the other objects is reliable and consistent. Consistency means that F is in front of another object iff decal is in front of that object.
  (iii) The pixel coverage of decal is a subset of the pixel coverage of F.

  Question: how would you modify the solution if (iii) fails?

§6. Stencil Buffer: Filling Non-Convex Polygons

- `OpenGL` only guarantees that convex polygons are correctly filled. If we need to fill an ARBITRARY polygon, one solution is to use GLU's tessalation facilities (see below). But if the polygon is simple (but non-convex), then we provide a solution using the stencil buffer here.

- Note that in general, the filling of an arbitrary polygon P is not well-defined.

DRAW SOME CLOSED POLYGONS EXAMPLES: A SPIRAL CURVE, TREFOIL CURVE, A SPRING (HELICAL) CURVE, ETC.

But we can proceed as follows. Pick an arbitrary point O (possible a vertex of P). Then for each edge of P (including the edges incident on O), we form a triangle. Call these the **O-triangles**. Relative to the set of O-triangles, every point in the plane now has a **covering number** which is the number of triangles it is contained in. This covering number is dependent on O (consider the case where O is slightly on side of an edge, versus slightly on the other side).

- PARITY THEOREM: *The parity (i.e., oddness or evenness) of the covering number of any point q in the plane are independent of the choice of O, provided $q \neq O$.* The proof is given below.

- It is not hard to see that if q is in the interior of a simple polygon P, then the parity of q is odd. We now define our polygon fill task as follows: given a closed polygon P, we want to fill in those regions of odd parity.

- The solution is as follows:
(1) Clear the stencil buffer with a zero value.
(2) Disable writing of color buffer.
(3) Set the stencil operation to `GL_INVERT` whenever the depth test and stencil test is passed.
(4) Fix any O and draw each O-triangle. HINT: `glBegin(GL_TRIANGLE_FAN)` mode is most efficient for this.
(5) Draw each O-triangle again, this time setting the stencil test to pass whenever the stencil value is non-zero.

- Why does this work? For each stencil pixel, its value will be inverted each time it is covered by a drawn triangle. The inversion is bit-wise (since the starting value is all bits turned off, the non-zero value simply corresponds to all bits turned on). Thus a non-zero value corresponds to an odd parity.

- Details of Stencil Operation: there are two stencil methods we need to invoke. Using

```
glStencilFunc(GLenum func, GLint refVal, GLuint bitMask),
```

we specify the stencil test. This test invokes a comparison function denoted by `func`. The comparison is between the MASKED reference value `refVal` and the MASKED current stencil pixel. By MASKED we mean that the respective values are first AND'ed with the mask `bitMask`. The possible values of `func` are `GL_NEVER, GL_ALWAYS, GL_LESS, GL_LEQUAL, GL_NOTEQUAL,` etc. For our current application, we set `func = GL_ALWAYS`; the other two arguments are irrelevant.

- The other stencil method tells OpenGL how to modify the stencil pixel after the stencil test:

```
glStencilOp(GLenum fail, GLenum zfail, GLenum zpass),
```

The possible values of each of these 3 arguments are `GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, GL_INVERT.` The last one is what we need here, and it performs a bitwise inversion of the current stencil pixel. The function `fail` is invoked if the stencil test (from previous bullet) fails. Otherwise, we invoke `zfail` or `zpass`, depending on whether the depth test fails or passes. Interestingly, we see that stencil operation depends on the depth test as well! For our current application, we set `zpass = GL_INVERT` and `zfail = GL_KEEP`. The argument `fail` is irrelevant.

§7. Tesselation

- GLU provides a facility to tesselate polygons. The implementation details are hidden – but what we are guaranteed is that an arbitry polyton is subdivided into a collection of convex polytons, so that filling in the tessation will lead to a filling in of the original polygon. Moreover, the definition of what is inside a polygon can be based of various tests on the winding number of a point. This gives a more general solution to the filling in of polygons than our Stencil Buffer solution above.

- The basic object is called `GLUtesselator`, which we can think of as a collection of convex polygons. To initialize such an object, we call `gluNewTess(void)` which returns a pointer to a `GLUtesselator` object. This should be the first command:

```
GLUtesselator * tobj = gluNewTess();
```

- We have analogues of the `glBegin()`, `glVertex()`, `glEnd()` blocks, but now there is a 2-level nesting of begin-ends:

```
gluTessBeginPolygon(tobj, userData);
gluTessBeginContour(toj);
⋮
glTessVertex(tobj, coords3d, vertexData); ⋮
gluTessEndContour(toj);
gluTessEndPolygon(tobj);
```

  Each “polygon” is really a polygonal region. A polygonal region is a connected set whose boundary components are called “contours”. Hence, within the `gluTessBeginPolygon` and `gluTessEndPolygon` block, there can be one or more contour block.

- Because of tesselation objects, a single OpenGL polygon may actually be a part of a larger polygon. Hence, when we want to draw the outline of the larger polygon, we need to know whether each edge of the constituent polygons are “interior” or “exterior”. Only the latter should be drawn in the outline. OpenGL provides a method to mark each polygon edge of with a Boolean value, called its “edge flag”. The default value of this edge flag is `GL_TRUE` (meaning “exterior”). To specify this value, you can place the following command

```
glEdgeFlag*( GLboolean eflag);
```

  anywhere inside a `glBegin(); glEnd();` block. Any subsequent vertices will be marked with with this flag, and this means is that the edge flag of the edge which starts from this vertex has the value `eflag`.

§8. Proof of the Parity Theorem

- Proof of the Parity Theorem: We say that O is **regular** if each O-triangle is non-degenerate. Consider any continuous motion of O that is **regular**, that is, for any line L through a vertex of P, the trajectory of O passes through L only a finite number of times. In particular, O passes through a finite number of irregular position. Pick any point q different from a vertex of P. The covering number of q changes whenever some side $[v, O]$ of an O-triangle passes through q (v is a vertex of P). But this happens when O passing through the line L containing $[v, q]$; by the regularity of the motion, O crosses the line L transversally. Consider the two O-triangles $\Delta_u = \Delta(O, u, v)$ and $\Delta_w = \Delta(O, v, w)$ where v, w are the two vertices of P adjacent to v. There are 3 cases: (0) u and w do not lie on L. Then the covering of q by Δ_u and Δ_w can change as follows: from $(0, 0)$ to $(1, 1)$ or vice-versa, or from $(1, 0)$ to $(0, 1)$ or vice-versa. But in any case, the parity of the contributions by these 2 triangles are unchanged. Now, the line L contain more than one vertex of P (we only assumed that v is on L). Our preceding argument already admit the possibility that w or v may lie on L. HERE WE ASSUME THAT NO TWO ADJACENT EDGES OF P ARE COLLINEAR. For any other vertex v' (different from u, v, w) that lies on L, we can repeat the argument, and again exclude its adjacent vertices u', w'. The conclusion is that the total contributions of the O-triangles incident on v, v', v'', etc to the parity of q is unchanged.

Since no other O-triangles change their contribution to the covering number of q, this proves our result. Q.E.D.

- For points q far enough from O and P, the covering number is even. For points arbitrarily close to a vertex that lies on the convex hull of P, we can find points that has odd parity. We may therefore define the **interior** of P to be those points with odd parity. In case P is a simple polygon, it is seen that this is the correct definition. For a non-simple polygon, other reasonable notions of "inside" is possible.

- There is a subtle point to be discussed. So far, we have deliberately not indicated whether points on the boundary of our triangles are considered inside or outside the triangle. But whatever the choice, the parity theorem is unchanged. But this could affect what is consider INSIDE or OUTSIDE P. For instance, what is the covering number of O? What is the covering number of points on an edge of P? We leave it to the student to explore these consequences.

- To finesse these questions, we offer the following definition of "insideness" for a given polyton P: an arbitrary point q is **inside** P if, relative to any choice of origin O, there are arbitrarily close points to q with odd parity.

§9. Pixmap Reading and Drawing

- There are three basic commands to move and copy pixels from or to a buffer:

- `glReadPixels()`: reads a rectangular array of pixels from framebuffer to memory

- `glDrawPixels()`: inverse of `glReadPixels()`. The position in the framebuffer depends on the "current raster position" (see below).

- `glCopyPixels()`: copies a rectangular array of pixels from one part of framebuffer to another. It is similar to a glReadPixels() followed by a glDrawPixels(), but without any data stored in memory.

- The read, draw and copy pixel commands has a frame buffer as its implicit argument. [NOTE: usually, the converse of "read" is "write", but in this context, the converse is called "draw".]

- To select one of the buffers to read from, we use `glReadBuffer(GLenum mode)` where `mode` is one of the following: `GL_FRONT`, `GL_BACK`, etc, and also `GL_AUX`i where i is some small integer.

- To select one or more of these buffers to draw to, we invoke `glDrawBuffer(GLenum mode)` where `mode` is one of those allowed for `glReadBuffer()`, but with two addional modes: `GL_FRONT_AND_BACK` and `GL_NONE`.

  ASIDES: (1) We can select only one buffer to read from, but can select more than one buffers to write to. (2) We can in fact select NO buffers to write to using `GL_NONE`. Interesting question: Why would we want to select no buffers? We already saw an example above!

- The concept of **current raster position** is also necessary to understand drawing or copying pixels. This is a position in world coordinates that is specified by `glRasterPos*()`.

- As far as the graphics pipeline is concerned, `glRasterPos*()` is treated exactly as a `glVertex*()` command until the point when rasterization begins. At this moment, the transformed 2-D coordinates for `glRasterPos*()` are saved.

§10. Pixel Formats and Types

- We now attend to the arguments of the pixmap/buffer operations. What is confusing is the fact that there are 3 independent families of parameters that govern what and how the pixel data in the color buffer is represented! We will discuss two of these ("format" and "type") now. The third family is called "storage mode" and is treated below.
- Let us begin by specifying looking at the arguments of the reading command:

  ```
  glReadPixels(x, y, w, h, format, type, *pixmap).
  ```

  We want to read a rectangle of pixels from the (implicit) selected color buffer into memory storage (which is specified by the pointer `pixmap`). The first 4 arguments are clear: the rectangle has origin (lower-right corner) at buffer coordinates (x, y) of width w and height h. The argument `pixmap`) points to an array of the proper units (8-bit, 32-bit, etc) of the correct dimension.
- What about the argument `format`? It has one of the values `GL_RGB`, `GL_RGBA`, `GL_BGR`, `GL_BLUE`, `GL_GREEN`, etc. So this "format" might be better called "color component".
- What about `type`? It has one of the values `GL_BYTE`, `GL_BITMAP`, `GL_SIGNED_BYTE`, `GL_UNSIGNED_BYTE_3_3_2`, `GL_UNSIGNED_BYTE_2_3_3_REV`, etc. In some cases above, in place of `BYTE` we may have `SHORT`, `INT` or `FLOAT`. We may sometimes be able to replace `SIGNED` with `UNSIGNED`, etc. Thus this "type" might be better called "bit distribution".
- Now we come to drawing:

  ```
  glDrawPixels(w, h, format, type, *pixmap).
  ```

- This is draw a pixmap (specified by `pixmap` into the (implicit) buffers selected for drawing. Compared to reading, we have lost two arguments (x, y). Why? That is because (x, y) is determined by the current raster position! If the current raster position is invalid, nothing is drawn.
- Finally consider the copying instruction:

  ```
  glCopyPixes(GLint x, GLint y, GLSizei wid, GLSizei ht, GLenum buff),
  ```

  This moves data from the buffer selected for reading (by `glReadBuffer()`), to the buffer(s) selected for writing (by `glDrawBuffer()`).
- We illustrate the use of pixmap reading and drawing commands with the program pixmap.cc adapted from Hill's book.

§11. Pixel Storage Mode

- Both `glPixelRead()` and `glPixelDraw()` are affected by a family of parameters called "pixel storage modes".

```
glStorei(GLenum PARAM, int PARAM_VAL),
glStoref(GLenum PARAM, float PARAM_VAL).
```

Examples of PARAM are GL_PACK_SWAP_BYTES and GL_UNPACK_SKIP_ROWS. The value of the former is a boolean, and the latter is an integer. However, we can use integer values to specify the value of GL_PACK_SWAP_BYTES: 0 for false value, and non-zero for true value.

- These parameters can be classified as either (1) packing modes or (2) unpacking modes. The two PARAM's above illustrate the syntax for both classification.

  Packing modes determine how pixmap data is stored into memory. They are therefore relevant to glPixelRead().

  Unpacking modes determine how pixmap data are retrieved from memory. They are therefore relevant to glPixelDraw() and glBitMap().

  Other affected commands relates to texture operations such as glTexImage*().

- In the following, we omit the packing/unpacking classification in our parameters: thus we write *SWAP_BYTES to refer to both GL_PACK_SWAP_BYTES and GL_UNPACK_SWAP_BYTES.
- There is a family of pixel storage modes that control byte alignment, byte and bit order. For instance, if the *SWAP_BYTES parameter is equal to GL_TRUE, it means that the transfer of any pixel data to/from memory will have their bytes being swapped depending on whether we have set the packing/unpacking modes.
- We can skip the selection of byte alignment parameters if we are operating within the context of one platform, and let the default parameter values operate.
- Here, we focus on three important storage mode parameters if we want to draw subimages. This ability will be critical for implementing the panning operation.

```
int wid = 0;
glPixelStorei(GL_UNPACK_ROW_LENGTH, wid);
int offset_x = 0;
glPixelStorei(GL_UNPACK_SKIP_PIXELS, offset_x);
int offset_y = 0;
glPixelStorei(GL_UNPACK_SKIP_ROWS, offset_y);
```

  In the command glPixelDraw(w, h, ...), the default assumption is that the pixmap has w columns and h rows. But in case we are trying to draw a subimage in the pixmap, we need to specify the actual number of columns in each row, using the GL_UNPACK_ROW_LENGTH parameter. The other two parameters tells OpenGL what is the x- and y-offsets for the lower-left corner of the subimage.

- We illustrate the use of pixel storage modes in a program for panning images: pan.cc .

§12. BitMaps

- Bitmaps are a special case of pixmaps where each pixel is a bit. There is special command for drawing bitmaps:

```
glColor3f(1.0, 1.0, 1.0); // white
glColor3f(1.0, 1.0, 1.0); // set raster position  1
glColor3f(1.0, 1.0, 1.0); // red
glBitMap(...); // display the bitmap
```

- When we draw bitmaps, the current pen color is used. Recalled that this color is set with `glColor3f()`. There is a programming subtlety here. Consider this example from the Red Book:

```
glColor3f(1.0, 1.0, 1.0); // white
glColor3f(1.0, 1.0, 1.0); // set raster position  1
glColor3f(1.0, 1.0, 1.0); // red
glBitMap(...); // display the bitmap
```

  Is the bitmap displayed in white or red? Answer is white. The red color only become be effective the next time we set raster position.

THE END