OpenGL Notes

Jeff Pferdehirt
jrp13@pitt.edu
http://www.sis.pitt.edu/~jeffreyp/is2780
7/16/2008

Blending

- We have already been dealing with "blending" to some degree already.
  - Recall that the A in RGBA stands for alpha.
    - We have seen RGBA with glColor*(), glClearColor(), Lighting parameters, material properties, and in many other places, already!
  - Alpha indicates how opaque the color of an object is (0 is 100% transparent, 1 is 100% opaque).
  - By manipulating the alpha component of an RGBA color, we are using blending.
- Recall that
  - Pixels on a monitor emit red, green and blue light.
  - The color of the pixels is determined by red, green and blue color values.
  - We cannot make the *screen* less opaque.
  - How, then, does the alpha value affect what is drawn to the screen?

Blending

- We can enable/disable blending using
  - glEnable(GL_BLEND) / glDisable(GL_BLEND)
- When blending is *not* enabled
  - Each new fragment overwrites any existing color values in the frame buffer.
    - This give the appearance of opacity.
  - The alpha value is ignored.
- When blending is enabled
  - Alpha values are most often used to *combine* the color values of a *fragment being processed* with that of the pixel *already stored in the frame buffer*.
    - This can give the illusion of transparancy.
- Blending occurs after the screen has been rasterized and converted to *fragments*, but *before* the final pixels are drawn to the framebuffer.

Blending

- Blending achieves the illusion of transparancy by combining the appearance of two or more objects as of function of their alpha components.

Blending

- During blending
  - The color value of the incoming fragment (the *source*) are combined with the color values of the currently stored pixel (the *destination*) in a two stage process.
    - First, the source and destination factor must be computed
    - After we compute these factors, we can them combine the two color values.
  - These entire operation can be summarized as follows

$(S_r, S_g, S_b, S_a) \rightarrow$ Source factors.
$(D_r, D_g, D_b, D_a) \rightarrow$ Destination factors.
$(R_sS_r+R_dD_r, G_sS_g+G_dD_g, B_sS_b+B_dD_b, A_sS_a+A_dD_a)$

Blending

- We can specify the generation method for the source and destination factors using one of the following functions
  - glBlendFunc(GLenum srcfactor, GLenum destfactor);
    - Choose two blending factors. The first for the source RGBA, and second for the destination RGBA.
    - *srcfactor* – multiplies the source RGBA value.
    - *Destfactor* – multiplies the destination RGBA value
  - glBlendFuncSeparate(GLenum srcRGB, GLenum destRGB
    GLenum srcAlpha, GLenum destAlpha);
    - Choose four blending factors.
    - Allows for the separation of color blending and alpha blending.

Blending

- The following table shows the possible values for BlendFunc and BlendFuncSeparate

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	destination	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(Rs, Gs, Bs, As)
GL_SRC_ALPHA	source or destination	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPH A	source or destination	(1, 1, 1, 1)-(As, As, As, As)
GL_DST_ALPHA	source or destination	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPH A	source or destination	(1, 1, 1, 1)-(Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

Blending

- In addition to specifying source and destination factors, we can specify a blending equation.
  - glBlendEquation(GLenum mode)
    - Specify a single blending equation.
  - glBlenEquationSeparate(GLenum RGB, GLenum alpha)
    - Specify a separate blending equation for RGB and alpha.
- By default, the blending operation calculates the sum of the source and default colors.

Blending

Blending Mode Parameter	Mathematical Operation
GL_FUNC_ADD	$C_sS + C_dD$
GL_FUNC_SUBTRACT	$C_sS - C_dD$
GL_FUNC_REVERSE_SUBTRACT	$C_dD – C_sS$
GL_MIN	min(C_sS, C_dD)
GL_MAX	Max(C_sS, C_dD)
GL_LOGIC_OP	C_s *op* C_d

Blending

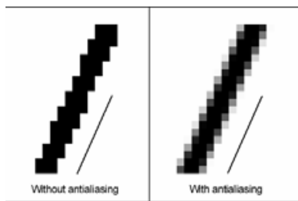
- The order in which a polygon is critical for blending operations (demo)
  - Consider a window with a sphere drawn behind it.
  - If we draw the window, and then the sphere, will the sphere still be seen?
    - No!
  - Why?
    - The window is closer to the eye than the sphere. Since the window is drawn first, it is already in the *depth buffer*.
    - When OpenGL goes to draw the sphere, it will determine that it is already occluded by the "window".
- This means that we must solve this problem manually *before* drawing our objects.

Blending

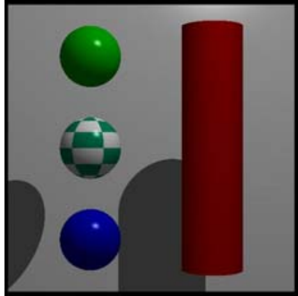
- How do we solve the drawing order problem?
  - For simple cases (with relatively few transparent primitives) we can draw all the opaque objects first, followed by the transparent objects.
  - If many primitives are transparent (recall that one "object" may have many primitives), we must depth sort all of the primitives and draw them farthest to nearest (to the eye)
    - This is basically the "painters algorithm".
    - A note of caution – depth sorting may be slow and computationally intensive for complex objects / scenes.
  - Several more advanced methods (e.g. *depth peeling*) can also be used to more rapidly and efficiently draw transparent primitives.

Antialiasing

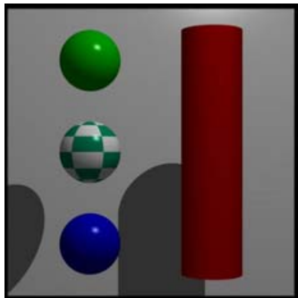
- You may have noticed that OpenGL will often produce lines appear jagged.
  - These "jaggies" appear because the ideal line is approximated by a series of pixels that must be located on a grid.
  - This jaggedness is called *aliasing*.



Antialiasing

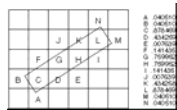


Antialiasing



Antialiasing

- When performing antialiasing
  - OpenGL calculates a *coverage* values for each fragment.
  - This value is based on the fraction of the pixel square on the screen that a fragment would cover
    - In RGBA mode, OpenGL multiplies the fragment's alpha value by its coverage.
    - This resulting alpha value is then used to blend the fragment with the corresponding pixel, already in the frame buffer.



- In general, antialiasing is computationally expensive.
  - glHint() can be used to exercise some control over the trade-off between image quality and speed.
  - Not all OpenGL implementations will "take the hint"

Antialiasing

- void glHint(GLenum target, GLenum hint)
  - Values for *hint* can be:
    - GL_FASTEST, GL_NICEST, GL_DONT_CARE

Parameter (for *target*)	**Specifics**
GL_POINT_SMOOTH_HINT, GL_LINE_SMOOTH_HINT, GL_POLYGON_SMOOTH_HINT	Sampling quality of points lines, or polygons during antialiasing operations.
GL_FOG_HINT	Whether fog calculations are done per pixel (GL_NICEST) or per vertex (GL_FASTEST)
GL_PERSPECTIVE_CORRECTION_HINT	Quality of color and texture coordinate interpolation.
GL_GENERATE_MIPMAP_HINT	Quality and performance of automatic mipmap level generation
GL_TEXTURE_COMPRESSION_HINT	Quality and performance of compressing texture images

Antialiasing

- Points and Lines
  - We can enable antialiasing for points and lines using glEnable() with either GL_POINT_SMOOTH or GL_LINE_SMOOTH.
  - In RGBA mode, we must also enable blending (recall how the *coverage* area is implemented).
    - Blending must be setup appropriately as well. glBlendFunc( ) can be used to alter the appearance of antialiasing.
- Polygons
  - "Enabling" general antialiasing is slightly more complicated.
  - General antialiasing is done using a technique called *multisampling*.
    - Normally, each fragment, has one color, depth and texture coordinate set, all based on the center of that fragment.
    - With multisampling, each fragment has *multiple* colors, depths, and texture coordinate sets, based on the number of sub-pixel samples.

Antialiasing

- Multisampling
  - Does not use the alpha value to perform antialiasing.
    - This means that depth sorting is not necessary with multisampling.
  - Multisampling also deals with traditionally difficult problems such as intersecting polygons.
- To enable multisampling, we must perform the following steps
  - 1. Obtain a window that supports multisampling
    - glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_MULTISAMPLE);
  - 2. Check that multisampling is enabled. If querying GL_SAMPLE_BUFFERS returns 1 and GL_SAMPLES returns greater than 1, you may use multisampling.
    - Glint bufs, samples;
    - glGetIntegerv(GL_SAMPLE_BUFFERS, &bufs);
    - glGetIntegerv(GL_SAMPLES, &samples);
  - 3. Turn on multisampling
    - glEnable(GL_MULTISAMPLE);

Antialiasing

- There are other methods of doing antialiasing
  - We can antialias polygons using alpha values as well
    - This method can be awkward and cumbersome.
  - We can also use the accumulation buffer to perform antialiasing on the entire scene
    - This can be computationally expensive.

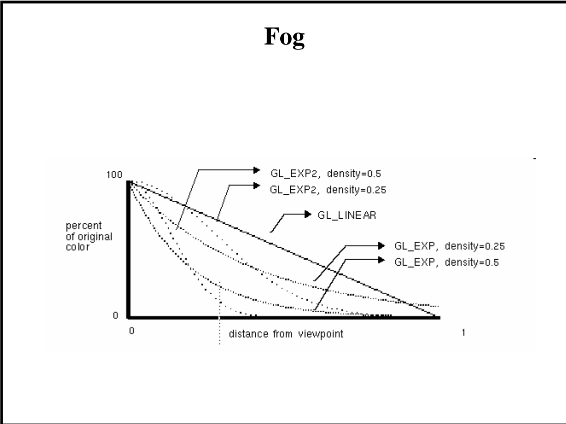
Fog

- Often, the images OpenGL generates are unrealistally sharp and well defined.
- Antialiasing aids a realistic appearance by smoothing object's edges.
- Fog is another used to increase the realistic look of OpenGL images
  - When fog is enabled, object that are farther from the viewpoint begin to fade into the fog color.
  - We can control the "density" of the fog – in general.
  - We can also use fog-coordinates (per vertex) for use in fog calculations.

Fog

- We enable fog using
  - glEnable(GL_FOG);
- We can further adjust the equation that controls the density/appearance of the fog using glFog*(GLenum pname, TYPE param);
  - If pname is GL_FOG_MODE param is:
    - GL_EXP – $f = e^{-(density*z)}$
    - GL_EXP2 – $f = e^{-(density*z)^2}$
    - GL_LINEAR – $f = {}^{end - z} /_{end - start}$
  - If pname is GL_FOG_DENSITY, GL_FOG_START or GL_FOG_END
    - param is a value for the respective parameter, clamped to [0, 1]
- We can adjust the appearance – performance ratio using
  - glHint( ) with GL_FOG_HINT

Fog



Fog

- Fog coordinates can be used, instead of the actual fragment location, for fog calculations.
- This gives a greater control over the appearance of fog.
  - In order to force fog calculation to use fog coordinates we can call
    - glFogi(GL_FOG_COORD_SRC, GL_FOG_COORD);
  - We can then specify the fog coordinate state glFogCoord*( )
    - Just like glNormal, etc, this sets the fog coordinate "state".

Animation

- Motion
  - One of the key components of animation is the appearance of smooth motion.
  - Traditional animation, like OpenGL, display a series of images, each slightly different from the last.
  - This give the appearance of "motion" to the objects within the scene.

Animation

- We can achieve similar effects in OpenGL by constructing a scene that changes slightly from frame to frame.
  - We have already seen this with our camera examples.
  - Consider how the camera was modified in then "backdrop" camera assignment.
    - The result was a scene that appeared to move.
- In order to properly animate an object, we must modify (slightly) and calculate several basic parameters of an object (or parts of that object).
  - Direction and location.
  - Change of position (velocity).
  - Rotation and scaling

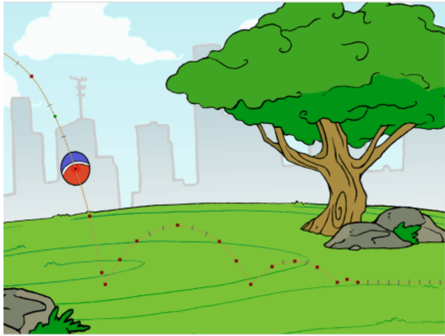
Animation

- Direction and location are key to animation
  - The location of an object is its position in space
  - The direction of an object specifies the vector along which an object is moving.
    - Modifying these give an object the appearance of moving through space.
- Rotation is also a key part of animation.
  - Determines the direction an object is facing.
  - May give an appearance of a "different" kind of "motion"
- Scaling may also be used.

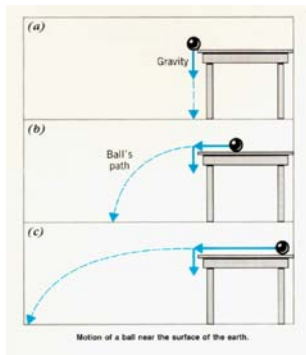
Animation

- Determining a path along which an object moves is one solution to calculating direction and location.
  - Using the path (with other parameters) we can determine where an object should be.
  - In many cases, simple linear movement is not enough to give a convincing representation of reality.
  - Parametric curves are often used as "paths"
- Another method is to calculate the object's parameters based on some physical type of movement.
  - Physical material properties
  - Gravity
  - Friction

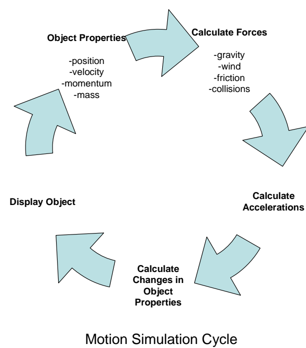
Animation



Animation



Animation

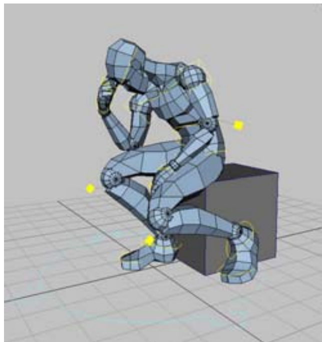


Motion Simulation Cycle

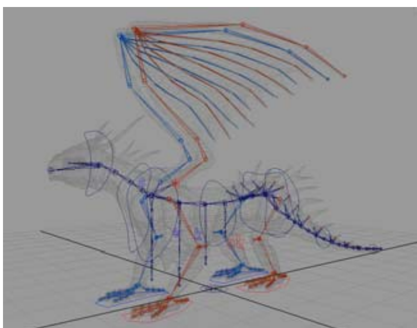
Animation

- This works well for simple objects (e.g., a ball) but what about more complex objects (e.g., a robot or human figure?)
  - In many more complex cases, an object must be broken into segments
    - Each of these segments has a series of properties, such as how it moves, directions it can be rotated, etc.
  - Groups of segments can also have properties
    - For example, consider a hand (as a segment) and a body (as a group).
    - The "body" as a whole has parameters.
    - The hand has its own parameters, independent of the body.
  - We can manipulate the properties of groups and segments.
  - We then draw the complex object by traversing down the object "tree" making the proper translations, rotations and scaling as we go.

Animation



Animation



Animation

- In general, this process of animating using an "object tree" is known as kinematics
- There are two general types of kinematics
  - Forward Kinematics (FK)
    - This is the process previously described.
    - Animation takes place starting at the root and working toward leaves
  - Inverse Kinematics (IK)
    - In inverse kinematics, the positions and rotations of leaves are set.
    - Using an algorithmic "solver", the positions and rotations of nodes further up the tree are calculated based on parameters.

Animation

- By altering the parameters slightly each frame we achieve a smooth animation.
  - It doesn't matter whether the parameters are FK parameters, IK parameters, physical parameters, etc.
- By combining certain relative change in parameters we can achieve specific types of contextual motion
  - When a car moves
    - It must be translated appropriately.
    - It may be rotated if turning.
    - The wheels must turn on their horizontal axis.
    - The wheels may turn about the vertical axis.
  - When a person walks
    - Their entire body must be translated.
    - Parts of the body may rotate around joints.
    - Their knees, arms, hands, and feet may move, rotate, etc.

Animation

- Demonstration – Atlantis
  - glutIdleFunc() → Animate() → SharkPilot() → Display() →

    FishTransform() → DrawShark()
- Demonstration – Mech
  - Here, each "joint" and "group" has parameters (position, rotation, scaling, etc).
  - We can modify these parameters directly
    - The result is that all nodes "down the tree" are affected.
  - We can combine a series of preset parameters to achieve an animation.
    - Walk cycle.

Readings

- Chapter 10 – Computer Graphics Theory into Practice
- OpenGL programming guide – Chapter 6 (optional)