CMPS 415/515 Fall 2006 Assignment 1

Due: Tuesday, Sept 12, 2006

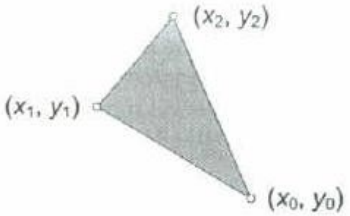
Source code must be submitted electronically at any time on Sep. 12 or earlier, *and* a printout must be submitted to the grader (printout is due when the grader arrives for office hours the next day). Note the "late policy" as given in the course syllabus.

Students may help each other understand concepts but may not share source code or submit source code from web sites, etc. Note the consequences given in the syllabus.

Assignment 1 is to implement scan conversion of a triangle into a simulated frame buffer. Triangle vertices will be specified by three user mouseclicks, and the triangle should be shaded smoothly to have three different colors at the vertices. A running example was shown in lecture.

Example programs on Moodle illustrate how a frame buffer can be simulated and how mouse button events can be handled. Extend these to store three successive mouse click coordinates and to call your scan conversion routine once three vertices have been specified. Do *not* use OpenGL commands for triangle drawing. Draw only by setting color values in the simulated frame buffer.

Your implementation should follow methods described in lecture, with the following high-level algorithm:



Given three vertices (x_0, y_0), (x_1, y_1), (x_2, y_2),
such that $y_0 \le y_1 \le y_2$, and
letting $L_{i,j}$ denote the edge from (x_i, y_i) to (x_j, y_j),

For each scan line from y_0 to y_1-1:
    Compute the two x-coords where $L_{0,1}$ and $L_{0,2}$
    intersect this scan line, and color pixels in this range

For each scan line from y_1 to y_2:
    Compute the two x-coords where $L_{1,2}$ and $L_{0,2}$
    intersect this scan line, and color pixels in this range

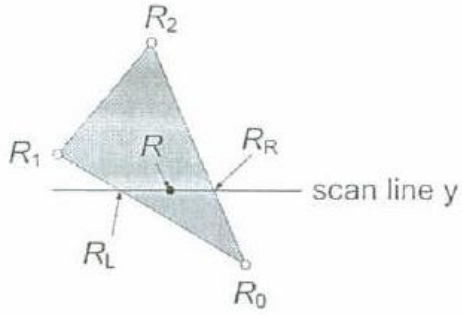
Some details are omitted by the above high-level algorithm:

- Your program should work for any 3 vertices. To guarantee that $y_0 \le y_1 \le y_2$, the three stored vertices must be passed to your scan conversion routine in the correct order, or sorted (it's usually best to not waste time with swaps).
- The calculation of intersection x-coords should be done efficiently, using an incremental approach like DDA. For example: at scan line y_0, $L_{0,1}$ intersects the scan line with x-coordinate x_0, so x_0 is the initial value for $L_{0,1}$ intersection. And, each time the scan moves up one line, the new x-coordinate of intersection can be found by adding $[(x_1 - x_0)/(y_1 - y_0)]$ to the previous value (this is a floating point ratio that should be precomputed, rather than once per iteration). $L_{0,2}$ and $L_{1,2}$ intersections can be handled by a similar method. At line y_1, the program stops tracking the $L_{0,1}$ intersection and initializes the $L_{1,2}$ intersection.

- Edge intersection coordinates are not necessarily integers. For a scan line, color the pixels from $\lceil x_L \rceil$ to $\lfloor x_R \rfloor$, where x_L and x_R are x-coordinates of the two intersections ($x_L \leq x_R$).
- If $y_0 = y_1$ or $y_1 = y_2$, there is a horizontal edge, and computing the above ratios could lead to a divide-by-zero condition. This can be checked easily. When $y_0 = y_1$, the first "half" of the algorithm is skipped. When $y_1 = y_2$, the second half should just draw one line.
- In the diagram above and in the discussion below, $L_{0,1}$ and $L_{1,2}$ are to the left of $L_{0,2}$. The program should handle both this case and cases with $L_{0,2}$ to the left of $L_{0,1}$ and $L_{1,2}$.

Coloring the triangle:

The scan conversion routine should be extended to smoothly interpolate color values from vertices. Suppose each vertex (x_i, y_i) is also given a color value (R_i, G_i, B_i) of your choosing. The diagram below helps illustrate an interpolation process that computes the red level R at a generated pixel. A similar process can be used to compute green and blue levels.



Suppose scan conversion has just generated the illustrated interior pixel, and call its coordinate (x, y). The color calculation first interpolates red levels along the two edges intersected by the current scan line to compute levels R_L and R_R. Then a final red level, R, is computed by interpolating between R_L and R_R. More specifically, for the illustrated case:

$R_L = s * R_1 + (1 - s) * R_0$, where $s = [(y - y_0) / (y_1 - y_0)]$,
$R_R = t * R_2 + (1 - t) * R_0$, where $t = [(y - y_0) / (y_2 - y_0)]$, and
$R = u * R_R + (1 - u) * R_L$, where $u = [(x - x_L) / (x_R - x_L)]$ and x_L, x_R are edge intersection coords.

But, for efficiency, these levels should be computed incrementally. For example, consider R_L. At scan line y_0, R_L is initialized to R_0. Each time scan conversion moves up one line, R_L can be computed incrementally by adding $[(R_1-R_0)/(y_1-y_0)]$ to its previous value. A similar process is used to compute R_R. This leaves R to be computed incrementally as a span of pixels is filled in.

As the span is filled, R is incremented by $\Delta_R = [(R_R - R_L)/(x_R - x_L)]$ for each rightward move. Recall that edge intersection coordinates x_L and x_R are not necessarily integers. So, R is not initialized to R_L. Instead, it is initialized to $[R_L + (\lceil x_L \rceil - x_L) * \Delta_R]$, to account for the distance between (x_L, y) and the actual left pixel coordinate $(\lceil x_L \rceil, y)$.

CMPS 415/515 Fall 2006, Programming Assignment 2

DUE: Sep. 28 via Moodle, plus a printout submitted to CC 438 by noon the next day.

Overview

The assignment is to draw a simple 3D helicopter in OpenGL (actually, to draw multiple instances of it) and to implement translation and rotation according to the methods below. Also, coordinate system axes must be drawn for the world and for each object. Further requirements will be given below and in lecture.

Helicopter Model

The helicopter model may be very simple, but it should have at least one closed nonplanar subcomponent without radial symmetry. Do not use any pre-existing models, model loaders, or rendering functions for this assignment. The local origin of the helicopter should be centered in its body – you will rotate about this point. Lighting is not required, but the coloring of polygons must allow the grader to clearly understand objects and their motions. You may want to keep in mind that a future assignment will require you to properly set surface normals for lighting.

Place helicopter drawing steps into a function to be called repeatedly in order to draw multiple helicopters. You will draw four helicopters, positioned in four different places, and rotating according to user input.

User Input

Using keyboard input, menu selection, or GLUI buttons, let the user request any one of 6 rotations: +X, -X, +Y, -Y, +Z, -Z. Any time an input is given, all four helicopters will rotate 15°, but the rotation will be interpreted and/or implemented differently for each helicopter. Rotation effects must be cumulative.

Required positions and rotations for the four helicopters

1) The first helicopter will be positioned at the world origin, will have orientation stored using X-Y-Z Euler angles, and three glRotate() calls will be used to set its modeling transform. This means you need to store three values to represent X, Y, and Z rotation angles, initially zero. For each input, a 15° increment will be added to the appropriate stored angle. The display function will then pass these angles to glRotate() calls along with descriptions of the corresponding axes. Note that, using our class conventions, calls such as these build the modeling matrix in "left-to-right" order. You will need to figure out which order of calls corresponds to X-Y-Z Euler angles.

2) The second helicopter will be positioned on the X-axis at some reasonable position (visible, and out of the way of the others). Its orientation will be stored as a 4×4 homogenous transform, rotations will always be about principal axes of its current local frame, and its modeling matrix will be built with one glTranslate() call and one glMultMatrix() call. This means you need to store its orientation as a matrix, initially identity, and multiply a 15° X, Y, or Z rotation matrix onto it for each input. You will need to figure out the correct multiplication order and the order for a glTranslate() call that positions the helicopter. When order is not correct, the helicopter may rotate about the world origin or only about world-aligned axes.

3) The third helicopter, like the second, will have its orientation stored as a matrix and updated using matrix multiplication. However, this one should be positioned down a different axis (whatever results in good visibility), should always rotate about world-aligned axes instead of local axes, and no glTranslate() call is allowed. In place of glTranslate(), helicopter position can be set directly in the fourth column of a homogeneous transform being passed to glMultMatrix(). However, this position should *not* be in the transform while computing a new orientation in response to input.

4) The fourth helicopter will appear to rotate just like the second, while its position will be set to form a rectangle with the other helicopter positions. However, the implementation will use quaternions in place of rotation matrices, and the modeling transform will be built using only one glRotate() call and one glTranslate() call. For this, a quaternion can be stored for current orientation. It is initialized to have a 1 in its scalar component and 0 in the others. In response to user input, one of six update quaternions is multiplied onto it (each of these six can be precomputed using the formula from lecture by plugging in an angle and one of the three principal axis descriptions). Depending on multiplication order, the helicopter can be made to rotate like either the second or third helicopter (the requirement is to match the second helicopter's rotations). When needed, the quaternion can be converted to angle-axis form, and the result passed in with a single glRotate() call.

Additional notes, hints, and clarifications

- For reasonable behavior, the changes made to the OpenGL modeling transform for one helicopter should not be applied to the next helicopter. There are two ways to handle this: reset the modelview matrix using glLoadIdentity() once per helicopter, or use glPush()/glPop() pairs to save and restore a previous state. Note that if you changed the view transform with a call such as gluLookAt(), then resetting to identity is not what you want (unless you want to set the camera pose on a per-helicopter basis). The modelview matrix combines modeling and viewing transforms.

- You may use only the following routines from an approved math library (GMTL or other code given on Moodle): matrix multiplication, quaternion multiplication, and quaternion-to-angle-axis conversion. All other matrix or quaternion manipulations should be handled directly by your own code (if in doubt, ask).

- Be careful to consider whether functions expect angles or degrees.

- For keyboard input, glutKeyboardFunc() registers a callback. Menus can be created using glutCreateMenu(), glutAddMenuEntry(), and glutAttachMenu(). A GLUT-based GUI can be created with GLUI.

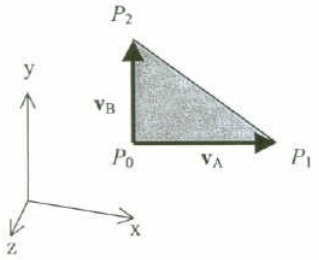
- Within a callback that handles input, you can update variables (stored orientation for each helicopter) and then use glutPostRedisplay() to cause a redraw. This causes GLUT to call your display function *after the current callback exits*. Avoid calling glutPostRedisplay() from within the display function itself, since this causes continuous redraws even when there is no need to redraw.

- If your objects are shearing or showing other strange warps, a matrix being passed to glMultMatrix() may need to be transposed. Or, for some other reason, you have something other than a rotation in the first three columns or something other than (0 0 0 1) in the last row.

- After many rotations, it is possible for error to accumulate in rotation matrices or quaternions, causing shearing or other strange artifacts. This can be fixed by reorthogonalization or renormalization, but you don't need to do this. Many rotations should be possible without problems.

CMPS 415/515, Fall 2006, Written Homework, Due Sep. 26

Consider the triangle below with vertices: $P_0 = \begin{bmatrix} 8 \\ 3 \\ 3 \end{bmatrix}, P_1 = \begin{bmatrix} 20 \\ 3 \\ -3 \end{bmatrix}, P_2 = \begin{bmatrix} 8 \\ 12 \\ 3 \end{bmatrix}$.



1) Compute the vectors $\mathbf{v}_A = P_1 - P_0$ and $\mathbf{v}_B = P_2 - P_0$ and their lengths.

2) Use a cross product of these to find a vector **n** that is normal (perpendicular) to the triangle surface. Specifically, find the normal pointing *toward* the reader.

3) Show that **n** is perpendicular to both $\mathbf{v}_A$ and $\mathbf{v}_B$ by computing dot products.

4) Normalize **n**, $\mathbf{v}_A$, and $\mathbf{v}_B$.

5) Assume coordinates are extended to homogeneous coordinates by adding the fourth coordinate W = 1. Give a homogeneous transform **T** for translating the triangle so the point $(\mathbf{T} \cdot P_1)$ lies on the origin.

6) Apply the transform to all three points (compute three translated points).

7) Describe a rotation that would make the bottom edge of the translated triangle fall on the Z axis, including exact rotation angle. If you are having difficulty, it may help to draw a top view of the environment (y pointed at you) and plot translated P_0 and P_1 on it, then find the angle between the Z axis and a line passing through the translated points (using trigonometry or linear algebra).

Express the rotation as:

a) a homogeneous transform (16 numbers),
b) X-Y-Z fixed axis angles (3 numbers),
c) an equivalent angle-axis description consisting of a unit rotation axis and an angle,
d) an exponential map (3 numbers), and
e) a quaternion (4 numbers).

8) Apply the rotation to the three translated points from question 6 and give the new coordinates.

9) What would have happened if the rotation had been applied before the translation? Give the exact coordinates that would have resulted.

CMPS 415/515 Fall 2006 Programming Assignment 3

Tentative due date for all parts: Oct 19

This assignment has more parts and will be more heavily weighted in grading than previous assignments (at most twice the normal weight).

Part I Overview: Extend the helicopter program to create a scene with two independently controlled helicopters with moving subparts and a movable perspective camera as detailed below.

Part II Overview: Improve visual quality of the scene by making effective use of OpenGL lighting, adding texture mapping, and using a simple shadow technique for drawing a shadow on a ground plane.

Part I Further Requirements

Using keypresses or some other input mechanism, allow the user to control base pose of each helicopter. More specifically, allow the user to rotate the helicopter about each of its current local axes and to translate the helicopter in its current "forward" direction.

Extend the helicopter model to include moving subparts, also controlled by user input, such as rotating and tilting rotors, opening doors, etc. In addition to the main body, a helicopter model must have at least 3 independently controllable subparts, at least 2 of them involving rotation. Subparts must be attached to the helicopter in such a way that a subpart's coordinate system does *not* share its origin with the parent (helicopter base) coordinate system. Each subpart must move in a meaningful way with at least one degree of freedom, controlled by user input. For example, one subpart could be a cockpit door that rotates about an appropriate hinge axis with keypresses. Subpart geometry may be simple, but include at least one nonplanar subpart.

Include an option to draw local coordinate systems of all subparts, in addition to the world and helicopter base coordinate systems.

Use a perspective projection matrix (glFrustum instead of glOrtho). And, for the viewing matrix in modelview, create a controllable camera that follows one of the helicopters. Specifically, the camera should translate with the helicopter but not rotate with it, so the helicopter always appears centered on the screen but rotates separately. Let the user rotate the camera about the helicopter with azimuth and elevation controls, and move closer/further from the helicopter. For this, the camera-with-respect-to-helicopter transform is $\mathbf{C} = \boldsymbol{R_a R_e T_d}$, where $\boldsymbol{T_d}$ is a z-translation, $\boldsymbol{R_e}$ is an x-rotation, and $\boldsymbol{R_a}$ is a y-rotation. This assumes the helicopter's local y-axis points up with respect to the model.

Part I Implementation Notes

Implementation should reflect traversal of a hierarchical structure as described in lecture (see "traversing a tree" notes). You are *not* required to build a general scene graph system. Instead, you may follow the example from the lecture to determine the corresponding sequence of calls that results in a hard-coded traversal. The grader will check that you are maintaining the modelview matrix as though a graph structure is being traversed, where each move along an edge adds a transform to the modelview matrix, and each move back undoes this transform. Use OpenGL's glPushMatrix() to store the modelview matrix before following an edge, and glPopMatrix() to restore the previous value when moving back.

Here is an example of transformation steps to add a moving subpart: Consider a cockpit door. One rotation angle needs to be stored to represent the current state of the door, and the program also needs to know the (constant) position of the door axis (with respect to the helicopter) to position it on the helicopter. Processing the door node of the graph could then include: 1) a glTranslate() to position the hatch axis on the helicopter, 2) glRotate() to set the hatch rotation, 3) hatch drawing, 4) moving back to the previous graph node (restoring previous matrix state). Or, a homogeneous transform for the rotation and translation could be applied with glMultMatrix() instead.

Here are two ways to attach the camera to a helicopter: 1) imagine a camera node is attached to the helicopter node, with graph traversal beginning at the camera node, and with transforms inverted where required as described in lecture, or 2) imagine the camera node is attached to the world node but with camera pose $\mathbf{C_2} = \mathbf{T}\cdot\mathbf{C}$, where $\mathbf{T}$ is the helicopter's pose, so the first thing placed on the modelview matrix is $(\mathbf{T}\cdot\mathbf{C})^{-1}$. In either case, when the helicopter body is drawn, the modelview will have value $(\mathbf{C})^{-1}$, and when the world axes are drawn, $(\mathbf{T}\cdot\mathbf{C})^{-1}$.

Part II Further Requirements:

To get credit for this part of the assignment, your scene must include at least some parts that were not created with the help of any modelers, model loaders, or even utility functions such as glutSolidCone(). At least one object in your scene must be a representation of a curved surface (smoothly shaded) for which you yourself have explicitly set vertex coordinates, texture coordinates, vertex normals, and material properties, *or* these may be generated by an algorithm written entirely by you for generating surfaces from an analytical model. In a comment at the top of your code, specify what technique you used to generate each object in your scene, and be very clear about any tools or existing data you used.

Add a point light source to the scene, enable lighting, and set object material properties with glMaterialfv(). Lighting also requires correct unit-length surface normals to be set at vertices. Do not dynamically compute vertex normals for each rendering pass – instead, precompute or pre-assign them for efficiency. Also for efficiency, avoid using GL_NORMALIZE when possible. At least one surface defined by you must use smooth-shaded polygons to approximate a curved surface, such that vertices of a polygon do not have identical normals.

Set material properties so that ambient, diffuse, and specular effects can be clearly observed in the default view using the lighting controls described in the next item. Do not simply set all materials to one color.

Draw a small object such as a sphere to indicate the position of the point light source. Attach the light to the world coordinate system and allow the user to interactively translate the light position along world x, y, and z directions. Allow ambient, specular, and diffuse light intensities to be switched on or off individually so a user can see their separate or combined effects as desired.

Color at least two surfaces in your scene with texture mapping, using at least two different images. Set texture parameters such that lighting effects remain visible on textured surfaces.

Draw a ground polygon (e.g., a horizontal quad) with projected helicopter shadows that are consistent with the current light position.

Part II Implementation Notes

Texture Mapping: A simple texture mapping program example with comments will be uploaded to Moodle. The source code will be named *tex.cpp* and the data file *Image.ppm* will also be needed. You should read all program comments and look up the OpenGL calls in a book or online (for example, in man pages, web pages, or in the Visual C++ help system). It is also recommended that you look at the Nate Robbins texture tutorial program to experiment with mapping parameters.

The example program includes a minimal image loader for ppm image files. To use images other than the provided sample, you can convert from other formats into ppm using free tools such as IrfanView on Windows or Gimp on Unix. For use with OpenGL code, images should be scaled to have widths and heights that are powers of 2. If you prefer to load a different image format directly, you may incorporate other image loading functions into your code as long as you properly credit their source.

You may use existing texture maps from outside sources as long as you clearly give credit to the source at the beginning of your program comments.

You may use existing utility functions provided that: 1) you credit the source and clearly identify the code segments used, and 2) these code segments only perform auxiliary functions and not the basic assigned requirements. For example, you may replace the ppm file loader from the texture example with a different image loading function. If in doubt, consult the instructor and always clearly credit the source.

Shadows: The required shadow technique is conceptually simple: Add a projection matrix to modelview that uses the light source as the projection point and the ground as the projection plane, and then redraw the helicopters in a shadow color.

You will be given the matrix for shadows on Moodle.

There are several ways to set shadow color. If the ground plane is a single color, then a simple technique is for the shadow color to be an ambient-only version of the ground plane color. If the ground plane is not one color (for example, it is textured with a ground image), then a common technique is to enable blending, disable lighting, and use the alpha channel to paint a somewhat transparent gray color over the current ground color to darken it. Other techniques exist for color, but fast techniques do not produce accurate shadow colors for nontrivial scenes because the color of real-world shadowed regions is influenced by high-order diffuse interreflections (as first described by Leonardo da Vinci) that are expensive to compute.

There are some visual artifacts you must prevent when drawing shadows. One is that the shadow appears at the same depth as the ground polygon, so round-off error may determine which one of the two is seen for any pixel. One solution is to place the shadow plane very slightly above the ground polygon. Another is to temporarily disable the depth buffer when the shadow is drawn (but then it should be drawn before any objects that are in front of it). Other artifacts can occur when a shadow is drawn with blending or when the shadow extends beyond the edges of a ground polygon (with blending, a pixel will have a different color depending on how many times color is added to it). OpenGL's stencil buffer can be used to make sure a single pixel is never colored more than once by the shadows, or to restrict the drawing region to the region covered by the ground polygon. More information will be given in lecture as needed.

CMPS 415/515 Fall 2006, Assignment 4, Part I

This is a preliminary description of assignment 4, with an expected due date of November 7. Part I is given here. Part II will be given in the next class meeting and has the same due date.

The assignment is to implement visible-surface ray casting (tracing) and color pixels according to an illumination equation. An arbitrary camera pose and perspective frustum should be supported, using parameters that are similar to those found in glFrustum and gluLookAt calls.

Additional work will be required from graduate students. Furthermore, graduate students who have successfully implemented a ray tracer in a past graphics class should show substantial new work (extensions) instead of resubmitting existing code.

Note that you will not use any OpenGL drawing or matrix functions, except display of a raster as in Assignment 1. The ray caster computes a color for each pixel and sets it in this raster.

Part I: Basic visible-surface ray casting

The pseudocode for a simple ray tracer is:

```
for (each scan line in image) {
    for (each pixel in scan line) {
        determine ray origin and direction for this pixel
        for (each object in scene) {
            if (object is intersected by ray and is closest considered so far)
                record intersection information, including object ID
        }
        set pixel's color according to the object at the closest intersection
        (or background color if no intersection)
    }
}
```

For the Part I description, assume an eyepoint, $\mathbf{p}_{\mathbf{eye}}$, at the origin (this will be generalized in Part II). Let the user provide parameters w, h, l, r, b, t, and n in an input file (a specific file format will be defined later), where $w \times h$ is the size of the desired raster in pixels and l, r, b, t, n are coordinates with the same meaning as the first five parameters of a glFrustum call.

To determine a ray origin and direction, note that origin is the eyepoint and direction is the direction from the eyepoint to a point, $\mathbf{p}_{\mathbf{pixel}}$, on the near plane viewing window that corresponds to the pixel currently being considered. Specifically, to map pixel coordinate (i, j) to a 3D point on this viewing window, use $\mathbf{p}_{\mathbf{pixel}} = (l + i * (r - l) / (w - 1),\ b + j * (t - b) / (h - 1),\ -n)$.

A ray originating from $\mathbf{p}_{\mathbf{eye}}$ and passing through $\mathbf{p}_{\mathbf{pixel}}$ can be described by the parametric line equation $\mathbf{p}(s) = \mathbf{p}_{\mathbf{eye}} + s\mathbf{d}$, where $\mathbf{d}$ is the ray's direction, $\mathbf{d} = \mathbf{p}_{\mathbf{pixel}} - \mathbf{p}_{\mathbf{eye}}$. Intersections between the ray represented this way and objects in the scene can be detected by routines in GMTL. Your ray caster should at least handle spheres and one other object type of your choosing. You will be given a sphere intersection routine that you may use in case you do not want to use GMTL. An attached excerpt from a textbook describes the derivation of this procedure.

When an intersection is found, the intersection routine should return the intersection coordinate or an s value that can be plugged back into the line equation to find the intersection point. Find the closest intersected object (if any) by performing ray/object intersection for all objects and then identifying the intersection with smallest s value or with the closest intersection point (direct use of s, if available, is more efficient).

After the closest intersection for a ray is found, set pixel color based on an illumination equation. The specific equation required will be given in Part II. For now, you can simply set the color according to the closest intersected object ID.

The resulting color should be stored for the pixel in a 2D raster as was done for setting pixels in Programming Assignment 1. Once all pixel colors have been generated this way, the resulting raster can be displayed with *glDrawPixels()*.

The scene description should be stored in a file that the grader can easily edit. You will be given a specific file format.

Ray/Sphere Intersection excerpt from Peter Shirley's *Realistic Ray Tracing*

NOTE: Shirley uses the symbol **o** for the ray's origin (above it is $\mathbf{p}_{\mathbf{eye}}$).

A sphere with center $\mathbf{c} = (c_x, c_y, c_z)$ and radius R can be represented by the implicit equation

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 - R^2 = 0.$$

We can write this same equation in vector form:

$$(\mathbf{p} - \mathbf{c}) \cdot (\mathbf{p} - \mathbf{c}) - R^2 = 0.$$

Again, any point $\mathbf{p}$ that satisfies this equation is on the sphere. If we plug points on the ray $\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}$ into this equation we can solve for the values of t on the ray that yield points on the sphere:

$$(\mathbf{o} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{o} + t\mathbf{d} - \mathbf{c}) - R^2 = 0.$$

Moving terms around yields

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c})t + (\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2 = 0.$$

Here, everything is known but the parameter t, so this is a classic quadratic equation in t, meaning it has the form

$$At^2 + Bt + C = 0.$$

The solution to this equation is

$$t = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

Here, the term under the square root sign, $B^2 - 4AC$, is called the *discriminant* and tells us how many real solutions there are. If the discriminant is negative, its square root is imaginary and there are no intersections between the sphere and the line. If the discriminant is positive, there are two solutions; one solution where the ray enters the sphere, and one where it leaves. If the discriminant is zero, the ray grazes the sphere touching it at exactly one point. Plugging in the actual terms for the sphere and eliminating the common factors of two, we get:

$$t = \frac{-\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}) \pm \sqrt{(\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}))^2 - (\mathbf{d} \cdot \mathbf{d})\left((\mathbf{o} - \mathbf{c}) \cdot (\mathbf{o} - \mathbf{c}) - R^2\right)}}{(\mathbf{d} \cdot \mathbf{d})}.$$

In an actual implementation, you should first check the value of the discriminant before computing other terms. If the sphere is used just as a bounding object for more complex objects, then we need only determine whether we hit it and checking the discriminant suffices.

The normal vector at point $\mathbf{p}$ is given by the gradient $\mathbf{n} = 2(\mathbf{p} - \mathbf{c})$. The unit normal is $(\mathbf{p} - \mathbf{c})/R$.

CMPS 415/515 Fall 2006 Final Project (Assignment 5), Part I

Due on the last class day

This describes Part I of a two-part assignment.

Part I: Implement a scene with moving particles, an object that attracts them, an object that emits them, and spherical obstacles that they bounce off of.

Part II will require scene elements to be moved along curve functions.

Summary of methods discussed in lecture

Storing particle state
Each particle can be stored as a six-element state vector **s** of 3D position **p** and 3D velocity **v**.

Creating particles
Initialize all particles to have some reasonable randomized **p**, **v**, and mass m. When a particle has moved very far from the scene or very close to the attractor, reset it at the emitter position with some reasonable randomized ejection velocity.

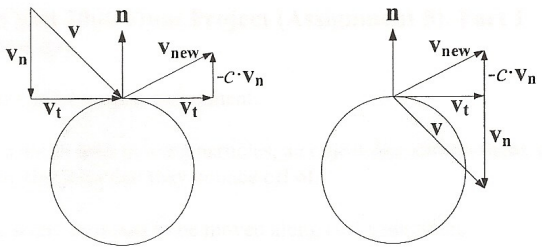
Particle motion
Let Δ be a time step (choose a constant Δ for whatever speed/accuracy you want). To advance an animation forward one step, update each particle with:

$$\mathbf{s} \leftarrow \mathbf{s} + \Delta \cdot \text{deriv}(\mathbf{s}), \quad \text{where deriv}(\mathbf{s}) \text{ is the derivative of } \mathbf{s}.$$

The elements of deriv(**s**) are **v** and **a**, where **v** is as above and **a** is 3D acceleration computed by Newton's second law, $\mathbf{a} = \mathbf{f}/m$, where m is particle mass and **f** is net force on the particle. Net force is the sum of attractor force ($k_1 m\mathbf{d} / r^2$) and viscous drag ($-k_2(\mathbf{v} - \mathbf{v}_{wind})$), where k_1 and k_2 are constants controlling effect strength, **d** is a *normalized* direction vector pointing from **p** to the attractor's position, r is the distance between **p** and the attractor, and $\mathbf{v}_{wind}$ is wind velocity.

Collision Detection
After **s** is updated as above, detect collisions with obstacles. Compute the distance between the particle and a sphere's center – if it is smaller than the sphere's radius, there is a collision.

Collision Response
When a collision occurs, bounce the particle by changing **v**. Let the sphere's surface normal at the collision point be **n**, computed using the difference between **p** and sphere center *and normalized*. Then $\mathbf{v}_n = [(\mathbf{v} \cdot \mathbf{n})\mathbf{n}]$ is the velocity component normal to the surface and $\mathbf{v}_t = (\mathbf{v} - \mathbf{v}_n)$ is the tangential component. Change velocity to $\mathbf{v}_{new} = \mathbf{v}_t - c \cdot \mathbf{v}_n$, where c is the restitution coefficient (the amount of elastic bounce, usually between 0.0 and 1.0). This ignores friction, since the tangent component is unchanged. Two ways of illustrating the components are shown:



Nonpenetration Constraints
There is a catch – when collision is detected this way, **p** is inside the sphere, not at the boundary. One fix is to move **p** just outside the boundary (e.g., to a point found by moving about one radius away from the sphere center in the **n** direction). Or, allow the particle to be in the sphere but only perform collision response when its velocity is actually inward, i.e., when $\mathbf{n} \cdot \mathbf{v}$ is negative. Otherwise, a particle on its way out can get stuck inside the sphere, especially for low c.

Drawing Particles
Draw a motion-blurred particle as a line segment from the particle's position before the state update to its position after the update. You may want to test with points before trying lines.

Animation in OpenGL
Use double buffering, a technique to be described in lecture. This is trivial – use GLUT_DOUBLE in place of GLUT_SINGLE and glutSwapBuffers() in place of glFlush.

One way to animate a scene is to use glutIdleFunc() (set_glutIdleFunc() if you use GLUI) during initialization to specify a function that gets called whenever GLUT has nothing else to do (this "idle function" gets called repeatedly when there are no events being handled). Your idle function updates the simulation by one step and then calls glutPostRedisplay(). The redisplay event is handled after the idle function exits, when GLUT processes the event by calling your display function. After the display function returns, GLUT calls the idle function again, and so on and so on. The result is that the idle function and the display function are called alternately, except that the call to the idle function can be delayed while GLUT processes other events such as keyboard input. This method preserves GLUT's event-handling capabilities.

Input
Allow user control of various parameters, including at least Δ, c, $\mathbf{v}_{wind}$, k_1, k_2, and some sort of control over obstacles.

Graduate students only:

Add a substantial effect or feature. If you have implemented similar work in a past class, you should show more substantial new work instead of resubmitting existing code.

CMPS 415/515 Fall 2006 Final Project (Assignment 5), Part II

Extension to basic particle system requirements

1) Limit each particle's lifetime by assigning a randomized integer at particle creation and decrementing it once per iteration until it reaches 0, at which time the particle should be reinitialized at the emitter. This will ensure there are always more particles to emit. (Note that using modulo when randomizing lifetime is appropriate, unlike for position and velocity, because lifetime is an integer. You will probably want to avoid small lifetimes).

2) Do something interesting with color or some other visual aspect of particles. There are no specific techniques required, to allow some freedom in developing effects, but credit will be deducted if only minimal (or no) effort is made. An example would be random minor color variations, color that varies meaningfully with properties such as age or velocity magnitude, use of blending, etc. Another example would be to use view-oriented billboards, but you should include an option to render particles as line segments.

Animating scene elements using Bézier curves

Animate scene elements other than particles by continuously varying them according to Bézier curves. Pick at least one 3D position, one other major 3D element, and one interesting system parameter to control this way (for example, emitter position, mean ejection velocity, and mean mass for new particles). Curve implementation must match the monomial form given in the course lectures and the construction below. No credit will be given for other implementations.

Suppose you are given at least three points $\mathbf{p}_i$, $0 \le i < n$. For each adjacent pair $(\mathbf{p}_i, \mathbf{p}_{(i+1)\%n})$, the midpoint between them is $\mathbf{m}_i = [(\mathbf{p}_i + \mathbf{p}_{(i+1)\%n}) / 2]$. A piecewise-cubic C^1-continuous curve can be passed through all of the midpoints by constructing a Bézier segment for each pair of adjacent midpoints. In the textbook's notation, a segment is $Q(t) = T \cdot M_B \cdot G_B$, $0 \le t \le 1$, where T is the vector $[\, t^3 \; t^2 \; t \; 1 \,]$, M_B is the 4×4 Bézier basis matrix, and G_B is the 4×3 geometry matrix with rows P_1, P_2, P_3, and P_4. Define the segment from $\mathbf{m}_i$ to $\mathbf{m}_{(i+1)\%n}$ to have a geometry matrix with rows $P_1 = \mathbf{m}_i$, $P_2 = (1/3)\mathbf{m}_i + (2/3)\mathbf{p}_{(i+1)\%n}$, $P_3 = (1/3)\,\mathbf{m}_{(i+1)\%n} + (2/3)\mathbf{p}_{(i+1)\%n}$, and $P_4 = \mathbf{m}_{(i+1)\%n}$. Animate by moving an object along this sequence of curve segments, looping around continuously, and give the user control over the t-increment used when animating.

The technique generalizes to other parameters: just grow the $\mathbf{p}_i$ to contain them ($\mathbf{m}_i$ and G_B grow accordingly). Or, repeat the above, but replace x, y, and z in $\mathbf{p}_i$ with the other parameters.

Read input from an ASCII file giving the n sets of values, one per line. Provide working sample input that represents at least six such sets, with variation in all degrees of freedom.

Draw some reasonable representation of inputs and curve constraints, and provide a mechanism for toggling this on and off. For example, for 3D position, draw small spheres at the input positions and at the four control points per curve segment as generated above. In case you are controlling velocity, inputs and constraints can be shown as vectors. In case of a parameter such as mass, a visual representation is not required, but effects must be clearly visible for credit.

515-level students only:
For Part II, use forward differences as described in section 11.2.9 of the textbook.