Lecture 14

Announcements

- Homework 5 CODE posted!
- Details and GitHub link on Piazza
- 0 points
- Free code for your team

New Demos

- View the bases used in a Scene as axes
- Text on 3D objects
- Multi-pass rendering
- Importing OBJ files
- Fast performance with tons of lights
- More Shapes (Parametric surfaces, surfaces of revolution)
- Stepping simulations, forces, collision testing

Parametric Surfaces

Arbitrary sheets of points, curved until they are geometrically closed

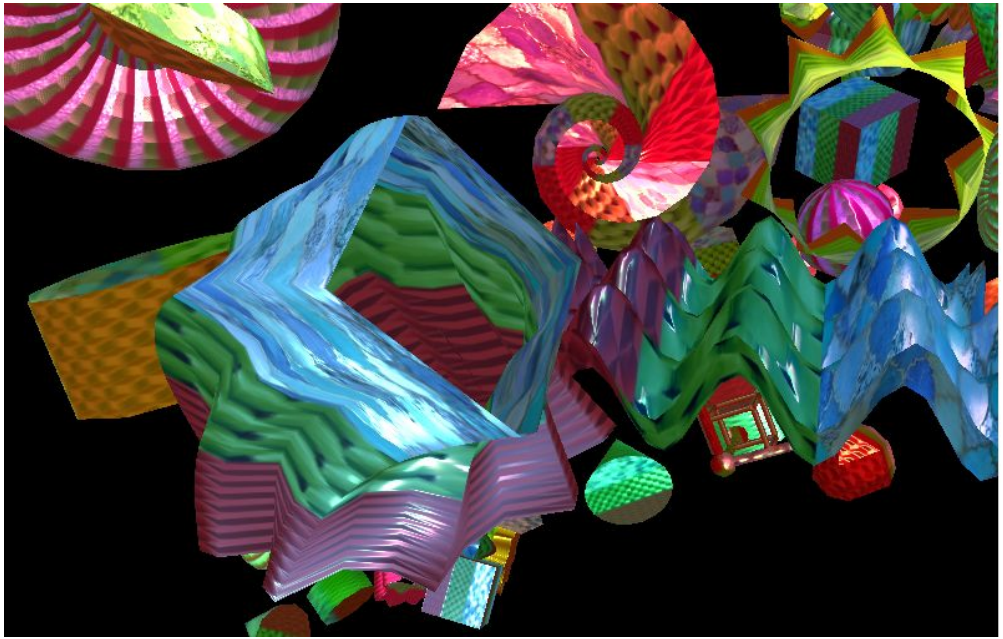
Closed Shapes

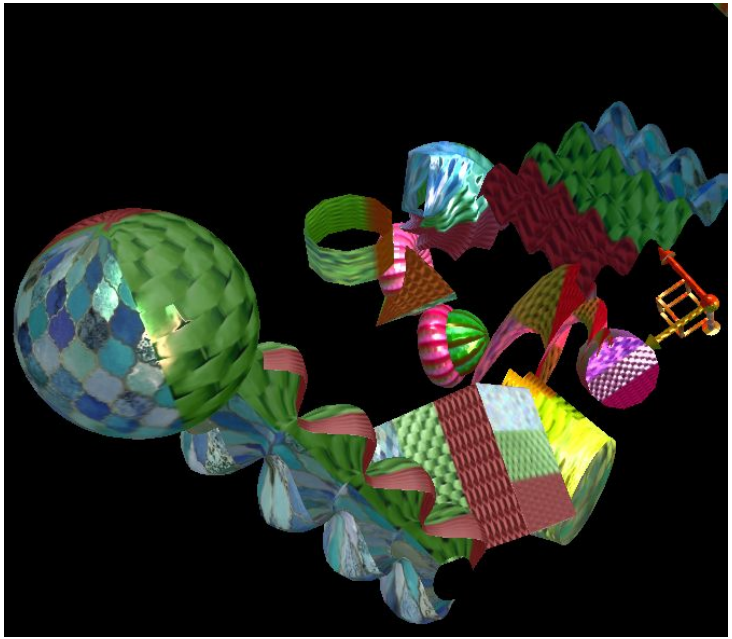
- Windmill is pretty but its geometry is not closed.
- Challenge: Make a closed, solid shape using those advanced practices.

- Surface_Of_Revolution in your code generates common closed shapes
  - Produces a curved “sheet” of triangles with rows and columns.
    - Begin with an input array of points, defining a 1D path curving through 3D space
    - Imagine each point is a row.
    - Sweep that whole curve around the Z axis in equal steps, stopping and storing new points along the way; imagine each step is a column.
    - Now we have a flexible "generalized cylinder" spanning an area

Grid_Patch

- This class drives the most complex shapes in your “shapes upgrade” demo called Surfaces_Demo.





Grid_Patch

```
class Cylindrical_Tube extends Surface_Of_Revolution    // An open tube shape with equally sized sections, pointing down Z locally.
  { constructor( rows, columns, texture_range ) { super( rows, columns, [ ...Vec.cast( [1, 0, .5], [1, 0, -.5] ) ], texture_range ); } }

class Cone_Tip extends Surface_Of_Revolution          // Note:  Curves that touch the Z axis degenerate from squares into triangles as they sweep around
  { constructor( rows, columns, texture_range ) { super( rows, columns, [ ...Vec.cast( [0, 0, 1],  [1, 0, -1]  ) ], texture_range ); } }
```

- Most of these shapes are made using tiny code due to the help of two classes: Grid_Patch and Surface_Of_Revolution (a special case of Grid_Patch). Grid_Patch works by generating a tesselation of triangles arranged in rows and columns, and produes a deformed grid by doing user-defined steps to reach the next row or column.

Grid_Patch

- A cone and cylinder are among the simplest and most useful new shapes.

- Also available is a set of axis arrows that can be drawn anytime that you want to check where and how long your current coordinate axes are. Draw it in a neutral color with the "rgb.jpg" texture image and the axes will become identifiable by color - XYZ maps to red, green, blue.

Grid_Patch

- All of these shapes are generated as a single vertex array each. Building them that way, even with shapes like the axis arrows that are compounded together out of many shapes, speeds up your graphics program considerably.

Custom Shapes

- Grid_Patch is your most flexible class for making shapes. For Project 2, you will get more mileage out of it than anything else if you use it creatively.
  - All you need to provide is a functions for reaching the next row and column from the current one.
  - Your functions will receive from Grid_Patch arguments of (i, p, j) where:
    - i is the progress through the rows (from 0 to 1),
    - j is the progress through the columns, and
    - p is the previous row or column's point.

- Surface_Of_Revolution can also yield a complex shape satisfying Project 2's custom polygon requirement rather easily.

Custom Shapes

- Example: A custom bullet shape can be made by simply storing some points in an array that follow the outline of a bullet, and using Surface_Of_Revolution to sweep that curve around the Z axis.

  - To make the outline of a bullet, keep pushing points onto an array. To generate the round parts, either use:
    - Loops and trig, or better yet:
    - Keep a “temp” point that you keep incrementally applying matrices to, stepping along the curve you want and adjusting the transform depending on how far along the bullet you are.

Particle Shaders

Video Tutorials

- https://www.youtube.com/watch?v=tfghiimtgyY&list=PLAwxTw4SYaPlaHwnoGxJE7NFhEWRCIyet&index=391

- https://www.youtube.com/watch?v=jsPfaQ7aMqk&list=PLAwxTw4SYaPlaHwnoGxJE7NFhEWRCIyet&index=499

Collision Detection

Testing for intersections to detect when your shapes are touching one another

Video Tutorial

- https://www.youtube.com/watch?v=ROOoj9t6-NA&list=PLAwxTw4SYaPlaHwnoGxJE7NFhEWRClyet&index=500

Motion

- Animated objects look more real when they have inertia and obey physical laws, instead of being driven by simple sinusoids / periodic functions.

Motion

- For each moving object, we need to store:
  - Model Matrix
  - Give the model transform matrix a "velocity" and track it over time
    - Need to split it up into angular and linear components so you can scale angular speed
- Example: For a bowling simulation, the ball and each pin would go into an array (including 11 total matrices)

Motion

- See the “inertia demo” for an example of using incremental motion to simulate momentum, velocity, and more.

Motion

- Next, see the “collision demo”.

Collision Detection

- Collision detection only means checking to see if your 3D, geometric volumes are touching in space.
- Usually done by checking pairs of volumes - ways to check fewer pairs exist but are a different topic.
- Collision resolution is what to do to the velocities afterwards - that's also a different topic.

Collision Detection

- Many queries in games can be solved with simple distance checks between objects - this is NOT usually called collision detection.
  - Checking if two chess pieces on a 2D board are touching
  - Checking if a character that only moves along the floor is within the walls of a map
- Collision detection is more of a 3D volume math problem.

Collision Detection

- The shapes we draw might be quite complex with a lot of edges sticking out (example: big .obj files)
- The math to exactly calculate whether two of them overlap could be intractable.
- In practice, graphics programs pretend complex shapes are replaced by simpler shapes during collision checks.
  - Collision boxes, collision cylinders, collision “capsules”
  - These enclose the real (drawn) shape.

Collision Detection

- Recommendation: Use collision ellipsoids.
- A stretched, rotated sphere can approximate your shapes about as well as a collision box can.
  - Example: Bowling pins would intersect correctly in most cases if we pretended they were just stretched ellipsoids, and not concave anywhere.

Collision Detection

- Which pins are in contact?
- Consider both position state and rotation state to find contact points

Collision Detection

- Ellipsoids degenerate into spheres in the right coordinate system.
- Math on spheres is easier to do than on things like boxes or cylinders – just test distance from the origin.
- Better than plain unstretched spheres
  - You'd like to be able to simulate bowling pins rotating or falling over and hitting each other, not just acting like a bunch of beach balls.

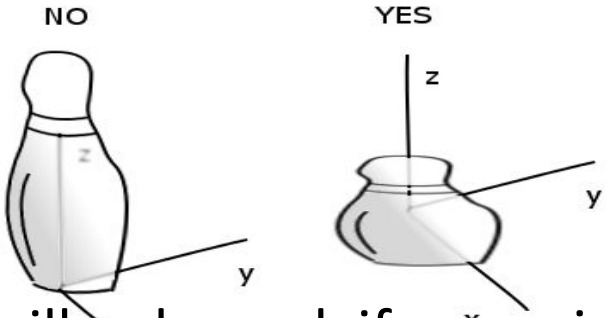
Collision Detection

- Each of your shapes has a model transform before you draw it.
- Just imagine using the same model transform on a unit sphere to mathematically represent a stretched out, rotated sphere there instead of your shape.
  - A much easier shape to work with, even though it's not what you're going to draw.

Collision Detection

- Warning: That sphere will only be a close approximation (tightly wrapping around your shape without missing too much) on the condition that:
  - Your object's vertices are centered around the origin and roughly unit length in all directions.

Collision Detection



- A bowling pin will only work if you originally define it as a very fat pin in its array of vertices, with the origin right at its center (not at the bottom or something).
- To make it a thin pin, the model transform is what is now in charge of that.

Implementation

- Make a big array of every object you want capable of colliding. Specifically, put their model transforms in a big array of mat4's.
- In our case, we have the Body class which does this for us; make an array of Body's.
- Loop through every pair, being careful not to collide a shape with itself:

for every transform $\mathbf{M_1}$ in the array:

for every transform $\mathbf{M_2}$ in the array such that $\mathbf{M_1}$!= $\mathbf{M_2}$:

Implementation

for every transform $\mathbf{M_1}$ in the array:

for every transform $\mathbf{M_2}$ in the array such that $\mathbf{M_1}$!= $\mathbf{M_2}$:

// In this loop, you have two transforms to compare,
// representing two possibly stretched out and rotated
// spheres.

// Remember:

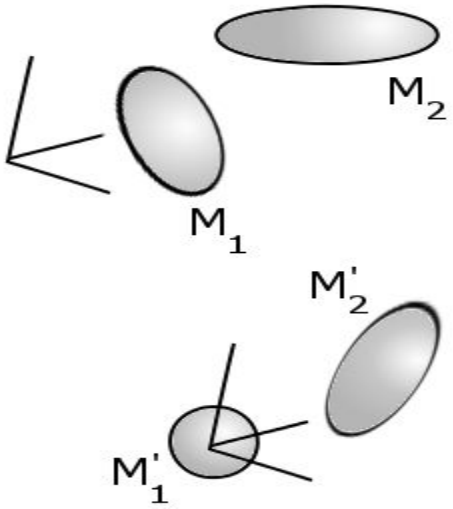
// Any drawn $Sphere_1$ is really just $\mathbf{M_1}$ times [each point of a unit sphere].
// Any drawn $Sphere_2$ is really just $\mathbf{M_2}$ times [each point of a unit sphere]

Collision Detection

- You know that, before being transformed, our collision ellipsoids were just unit spheres at the origin.
  - Hopefully that's what our untransformed model is also roughly shaped like.
- It's very easy to check if something collides with a unit sphere centered at the origin.
- If any part of it comes within a distance of 1 to the origin, it has to be inside such a sphere, right? Not bad.
- Let’s "undo" one of the transforms to get one of those ellipsoids back to being a unit sphere centered at the origin.

Collision Detection

- So, take M_1 inverse. What happens when you multiply that by each ellipsoid? Let's see:



- Multiplied by ellipsoid 1, that's $\mathbf{M_1^{-1}} * \mathbf{M_1} *$ (each point of a unit sphere), which cancels out, turning that one into the desired unit sphere shape.

- Multiplied by ellipsoid 2, that's $\mathbf{M_1^{-1}} * \mathbf{M_2} *$(each point of a unit sphere). This ellipsoid is just somewhere different now.

Collision Detection

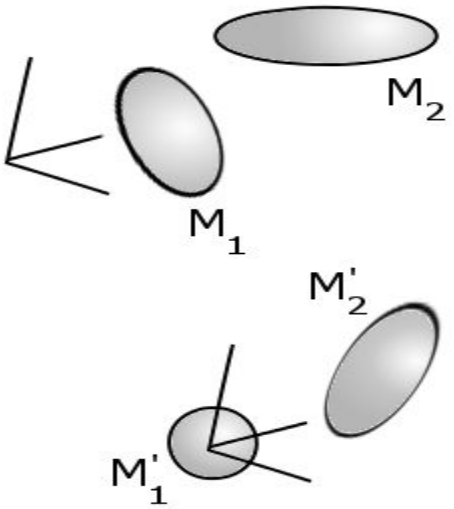
- So far we have code like this:

for every transform $\mathbf{M_1}$ in the array:

for every transform $\mathbf{M_2}$ in the array such that $\mathbf{M_1}$!= $\mathbf{M_2}$:

Let $\mathbf{T} = \mathbf{M_1}^{-1} * \mathbf{M_2}$

Collision Detection



- We undid one of the transforms and applied it to the whole universe, including our pair of volumes.
  - We stretch the universe in just the right way to make ellipsoid 1 line up with a unit sphere centered at the origin, dragging ellipsoid 2 along for the ride to somewhere else.
- Even though they both moved because of the change of coordinates, one thing hasn't changed: iff they were touching before -> they're still touching now.

That’s because affine transforms preserve certain phenomena.

Collision Detection

- Now we just need to check if our new Ellipsoid 2's position ever comes within one unit of the origin.
- Solving an ellipsoid-ellipsoid intersection is hard in math - a fifth order polynomial.
- Easier way: Just check points known to exist on the sphere, as transformed into our new ellipsoid. We have that in our Subdivision_Sphere's positions array.

Collision Detection

Final code: Take each point of our modified sphere 2 and see if it comes close enough to the origin (where sphere 1 is now):

for every transform $\mathbf{M_1}$ in the array:

for every transform $\mathbf{M_2}$ in the array such that $\mathbf{M_1}$!= $\mathbf{M_2}$:

Let **T** = inverse($\mathbf{M_1}$) * $\mathbf{M_2}$

For every point **p** of a unit sphere:

// (Just iterate through sphere.positions in your code)

Let **T_p** = **T** * **p**

if(length ((vec3)**T_p**) < 1)

{

// If we get here, the two shapes collide!!!

} // (Hopefully our ellipsoid approximation was close!)

Collision Detection

- What to do now, to react to the collision we may have found? Collision resolution / impulse
  - "Rigid Body Dynamics" studies this – part of physics

https://www.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf

https://www.cs.cmu.edu/~baraff/sigcourse/notesd2.pdf

You need your own quadratic matrix solver to handle resting contacts - that is not feasible. Instead, you'll have to settle for made up or random impulses when objects hit.

Collision Detection

- Note that our example lacks a collision hierarchy / bounding box hierarchy!
  - If you've only got a dozen objects that could collide at any given time, just checking every pair is fast enough.
  - When it's not enough, you implement a tree or hash table, CS32 constructs, to speed up the search through bounding shapes that contain one another.
    - Checking fewer candidate pairs that might intersect.