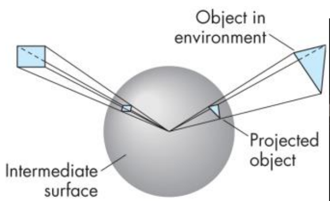
Final Study Guide – Nathan Tung

Lecture 7:

- Review: transparency order matters: draw over (render after) to show up on top (or flip for different blending)
- Environment Mapping (same a reflection mapping)
    - Reflection of a **fixed** environment; reflections computed via ray tracing (too slow in real-time environments)
    - Approximate the effect using texture map instead
        - Compute reflection vector (i.e. eye → surface → reflect → world); intersect that ray with scene and compute color (shading, actually) value; unfortunately, this is pretty much ray tracing
    - Approximate with two-pass method:
        - Place camera at mirror object location, facing in direction of mirror surface normal
        - Render scene without mirror into texture map
        - Render scene normally with texture map applied to mirror
        - This time with original camera position, mirror back in scene
        - Difficulties: where to put the camera? mirror is missing in first render (messes up light/mirror occludes something), projection from mirror can be tricky/off-axis, and we have to r-render texture every time camera OR mirror moves
    - How can we do environment mapping? Project scene onto sphere at COP
        - Viewer can't tell difference between object vs. projection (ex. Planetarium)
        - 


        - Given reflection vector, we find shaded color in scene and determine texture coordinates (s,t)
        - Not correct, but good enough (created at specific location like origin, not location of mirrored surface)
            - Need to re-compute for scene changes
            - Difficult but doable to create a map (real cameras like Google Street View use spherical lenses)
        - OpenGL is simpler: use projections, or even simpler, cube mapping!
            - For cube, render 6 images, each centered on an axis
            - Results are inside of a cube, unfolded; FOV must be 90 deg with matching edges though
            - Obviously, need to pass correctly transformed surface normal into vertex shader
            - Can compute reflection vector, pass into fragment shader (gets interpolated in fragment shader)
            - Notice: samplerCube type (similar to sampler2D) and textureCube function
- Bump Mapping (really displacement mapping)
    - Texture map stores displacements to the surface; requires computing normal at the new, displaced surface point
    - This needs partial derivatives to be solved; finite differences can be solved; slow when solving every fragment
    - Advantage is to use the result to perturb the normal in object space
    - Normal mapping: what we typically think of as bump mapping, but we're storing normal in texture map
        - X, y, z components of a normal vector are stored in the RGB channels of texture map
        - Need to map [-1,1] range of normals to [0,1] range of colors; use [R,G,B]=([x,y,z]+1)/2
        - Not enough: still need a new "coordinate frame" and "Tangent Space" (or TBN space)
            - Lighting for surfaces with normal maps applied are computed in TBN space
            - Tangent: tangent to surface at point p
            - Bi-tangent: cross product of the normal and tangent
            - Normal: surface normal at point p
        - TBN vectors define new frame at point we are lighting on surface; align T and B vectors with s and t dimensions of normal map; get object space to tangent space by a vector transformation
        - [Tx, Tz, Tz; Bx, By, Bz; Nx, Ny, Nz] ; transform light vector by this matrix
        - Light and normal map vectors are in same coordinate system, so lighting equations are correct

Lecture 8:

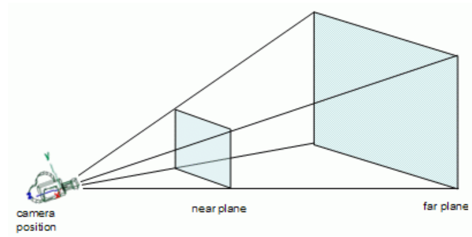
- Picking and Selection
    - Use mouse to select item; several methods available
        - Basic ones: old glSelect() mechanism; color buffer picking
        - Advanced ones: occlusion query extension; ray casting
    - glSelect() in OpenGL since v1.0, depreciated after v3.0, basically gone now
        - Very slow (implemented in software, stalls hardware rendering pipeline)
        - Idea is to use integers to "name" objects in the scene and mark stream of geometry with them

        - Works by rendering scene in special mode; buffer allocated to collect "hits"
            - A "hit" being any "name" intersecting view volume
            - gluPickMatrix() then helps restrict projection to area around mouse click (ex. 4x4 area of pixels)
            - Any primitives rendered after a "name" object defined is placed in predefined hit buffer
            - "Name" objects in hit buffer tell us what was under the mouse
    - Color Buffer Picking
        - Much simpler/faster; uses additional rendering pass
        - Every object we want to identify is given unique color (simple fragment shader, so directly assign color)
        - No lighting or texture mapping done; otherwise, scene is rendered normally
            - Z-buffer on, double buffering required (or rendered into dedicated color buffer for picking), only draw objects we wish to consider for picking
        - Once render pass is complete, DON'T swap buffers;
            - Call gl.readPixels() to recover result using mouse/window (x,y)
        - Pixel color value retrieved corresponds to object underneath the mouse
        - Variation of this technique uses separate color buffer rendered in parallel to normal buffer
        - Further enhancements for speeding it up:
            - Render only bounding volumes for scene objects
            - Be sure not to try/identify more objects than there are bits of precision in buffer
            - Only render objects that need to be selected
        - Problematic for objects with transparent textures (alpha values!), but manageable with some effort
    - Other techniques
        - Occlusion query – a special case
            - Want to know if something will be visible before we render it
            - Reading z-values instead of color
            - Allows us to determine order of objects under mouse
            - Harder to do region queries (e.g. rubber band box)
        - Ray Casting
            - Requires objects to be in memory (in some form)
            - Usually bounding solids used
            - Does not work at pixel level
            - Can have trouble with transparency (e.g. picking a hole)
- Collision Detection
    - Object Representation and Bounding Volumes, Simple Intersection Tests, Bounding Volume Hierarchies
        - Has object A collided with object B?
    - Object Representation
        - Compute whether one primitive intersects another
        - Could be done by comparing all triangles in a scene with one another, but slow/produces useless results
        - Stick only to objects in scene we care about; still checks objects not colliding (still bad!)
            - 10 objects with 100 triangles needs tens of thousands of comparisons
        - Better way: bounding volume
            - Sphere is simplest; not exact representation but only needs 50 comparisons
            - If two spheres intersect, check more closely (spend time only when needed)
            - Or test itself is enough if objects are far away enough and close inspection is unneeded
            - Too bad spheres aren't always a good choice for objects; good fit is desirable
        - Bounding rectangle can be better, but intersection tests more complex
            - Simplest box is an "axis-aligned bounding box" or AABB
            - Oriented bounding box can be best, but even more complex for intersection tests (tradeoffs!)
        - Bounding shapes affect accuracy/speed, but they all use distance d=sqrt((x2-x1)^2+(y2-y1)^2+(z2-z1)^2)
            - Ignore sqrt for speed!
        - Desirable Properties: inexpensive intersection tests (at least before restoring to expensive ones), tight fitting, inexpensive to compute BV, easy rotation/transformation, uses little memory
    - AABB
        - Simple to compute and straightforward!
        - Several ways to represent: Min/max extreme points (6 floats), min point and extends (6 floats or 3 floats + 3 half), center point and half-widths (same as above); last two are memory efficient
        - Computationally, min-extent is slowest and uses most operations
        - Intersection occurs if all axes overlap; if one axis is separated, no collision!

- Bounding sphere intersect if squared distance between centers <= squared sum of radii
  - Computing bounding sphere itself isn't so simple; brute force takes O(n^5)
  - Ritter uses iterative algorithm: start with sphere of 2 points, iteratively add points to grow (good for class)
    - Works quite well, but quality is sensitive to order of points added
  - Welzl uses computation geometry to achieve O(n) time, but complex implementation
- Processing BVs
  - Needs data structure
  - Organization depends on application
    - If we just need to check if spaceship hits asteroids, just check those (not asteroid-asteroid)
  - Advanced – what about hierarchies of bounding volumes?
    - If represented objects move, BV of bounding values have to update
  - Many bounding object types; spheres and AABB are less complex; maybe use sphere vs. plane (simple)

<u>Lecture 9:</u>

- Representing Models
  - Typical model rendering: make istance, set model view/shader, draw, repeat…
  - More objects to render makes this approach unmanageable
  - Objects are also usually related to one another (car has wheels stuck to chassis, chassis relation to road/light post)
  - Use **scene graphs** (or trees): objects can be represented in logical way
    - Object relationships set in hierarchical relation to other objects (ex. wheels related to chassis, not world)
  - The graphs/trees may look like TRANSFORM→CHASSIS→(TRANS→RFWHEEL, TRANS→LFWHEEL, etc…)
    - TRS (translate, rotate, scale) transformation in each transform node
    - To make entire car move, only need to move top layer
  - Processing scene graph
    - Traverse using DFS (depth first!)
    - Transformations processed like stack as we descend into tree (with variations on exact representation)
      - Push/save current transformation on way down
      - Pop/restore previous transformation on way up
    - May also want to track other application/graphics state in this way
  - Elements of scene graph
    - More flexibility to keep transformations separate from geometry
    - But we could fatten scene by coming transform nodes; or even apply transformations to static geometry
    - Node types: root, transform, object (instance: transform+object, grouped); also includes…
      - Group (common parent)
      - Attribute (set graphics state, ex. texture)
      - Predicates/switches (render particular child of node based on rule or index)
    - Each node type defines some predefined behavior, maybe callbacks for application-specific functionality
  - Root node: parent of all nodes in scene graph
    - Global state about scene kept here (e.g. lighting)
    - Storing camera info does NOT go here; scene graph represents scene alone
  - Group node: groups children nodes/graphs
    - Usually just a tree structuring element, or often a base class for many other node types
  - Transformation node: applies transformation to all child nodes
    - Usually based on group node
    - Two versions in some scene graph APIs
      - Dynamic (transforms change over time); static (no change, can be flattened for slow processors)
  - Object node: stores object geometry
    - Simple scene graph could store references to everything needed to draw
    - Often actually stores pointers to data structures representing data (vertices, colors, normal, texture coordinates, textures, shaders, etc.)
  - Most scene graphs also compute BV for each node in tree
    - Bounding sphere for all group based nodes is common
      - Root, group, transform, etc.
    - Object nodes with geometry usually use tighter-fit BV (used for culling scene graph or collision detection)
    - Culling determines which parts of scene are outside of current view fustrum
      - Only parts inside or partially inside fustrum are drawn (partially intersecting ones are drawn too)
      - Involves comparing BVs of scene with six planes making up the view fustrum

- View fustrum planes: 

  - Object completely inside or intersecting one or more planes must be rendered; ignore the rest
    - Need plane equations of all six sides of fustrum; extract planes directly from projection matrix!
    - Easiest to extract planes in clip space, where view volume (fustrum) is cube around origin
      - Coordinates are still homogenous coordinates
    - After applying MV and projection matrix, vertices are in homogenous clip coordinates
    - Normalizing performs perspective division, where view volume at (-1,1),(-1,1),(-1,1)
      - pc = (xc, yc, zc,wc) = PMp
      - pcn=(xc/wc, yc/wc, zc/wc)=(x',y',z')
    - But no access to values after perspective division (it's in hardware); but pc inside fustrum if…
      - Pc=(xc,yc,zc,wc)=PMp
      - –wc < xc, yc, and zc < wc
      - This means if –wc < xc, then pc is to the right of the left plane of the fustrum
      - Recall p=(x,y,z,1) and PM=[a11,a12,a13,a14; a21,a22,a23,a24;…;a41,a42,a43,a44]
    - If point p evaluated to 0, point is on plane; if >0, positive/inside; if <0, negative/outside
    - For intersecting with sphere, we need to normalize for distance values to mean something
    - Fustrum planes must be re-extracted when camera/objects in scene move (basically every frame)
    - Trivial to look up values for all 6 planes; as scene graph is traversed, BV is checked against fustrum; if it falls outside the fustrum, that branch of scene graph can be skipped

<u>Lecture 9.5 (Physics in TA session):</u>

- Particles, rigid body, cloth, water… (also deformable body, snow, smoke, fluid)
- We live in physical environment; realistic graphics is physical realism (how things move and look)
- Physics based animation
  - Particles, rigid body, deformable body, fluids, etc.
  - Real time vs. offline computation
- Physics based rendering
  - Global illumination, ray-tracing, path-tracing
  - Transparency, refraction, translucency (ios7), etc.
  - Depth of field
  - Motion blur
- Or just classical physics (no modern physics); things are getting to be real time with GPU power
- Particle Physics (trajectory of ball in angry birds)
  - For particle i: mi (mass), xi (position), vi (velocity)
  - f(t)=ma(t) = m(dv/dt)=m(d2x/dt2)
  - Numerical Schemes:
    - Forward/explicit Euler method: use current time step force and velocity
    - Backward/implicit Euler method: use future time step force and velocity
    - Big difference when force dependent on position of other particles (when position independent, no diff)
    - In code:
    - To start, initiate position, velocity, and force for each particle object
    - Per frame, calculate force (ex. constant gravity), used to update velocity, used to update position; draw
- Rigid Body (collapse in angry birds)
  - Three translational (chi) and rotation (theta) degrees of freedom
  - In code:
  - To start, initiate center of mass, moment of inertia, rotation, velocity, angular velocity, and force per rigid body
  - Per frame, calculate force/torque (no collision), update velocity/angular velocity, update position/rotation; draw
- Cloth, hair, 1D string; spring-damper system (slingshot in angry birds)
  - Newton's law of motion: mass*acceleration =force; ma=f; m(d2x/dt2)=f
  - System of second-order ordinary differential equations in time; net nodal force is f=s-gamma*v+g
  - -Gamma*v is nodal drag/damping force, g is gravity, s is spring force

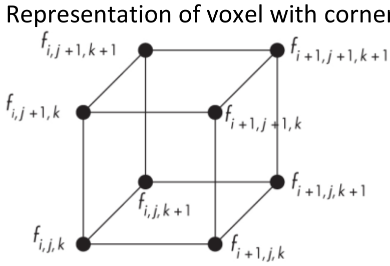
- Spring Force: net spring force at node i is the sum of forces due to springs connecting node i to neighboring nodes
  - S(t)=sum(Sij) for all j in neighbors; where Sij=cijeij*rij/|rij|;
    - rij=xj-xi, separation of nodes; |rij| is actual length; eij=|rij|-lij , deformation of spring
- A damped spring: parallel combination of spring and damper (Voigt model)
- In code:
- To start, init properties of nodes and spring-dampers
- Per frame, using forward euler: calculate sum of all forces, update velocity/position of nodes; draw everything

- Other forces
  - Friction force (stabilize the system): proportional to speed of node; friction factor
  - External force (manipulation): must solve pick problem; direct add to total force on picked node
- Advanced Deformable Object Simulation
  - Achieved by solving systems of partial differential equations
  - Continuum mechanics define equations
  - Apply advanced numerical methods to solve equations (Finite Element Method)
- Fluid and Smoke simulation
  - Navier-Stokes Equations; solved by applied mathematics
- Physics on the web?
  - How GPU shader-based simulation works
    - Fast, 10x improvement, no CPU resources used
    - GPGPU: general purpose computing on GPU (old way: programmable shaders; new way: CUDA/OpenCL)
    - WebGL? Supported on Chrome/Firefox by default; WebCL? Under development, but will be better
    - Shader:
      - Pipeline iterates at >30 fps; vertex and fragment shaders execute in parallel on many GPUs
    - Simulation shaders have two problems:
      - First, set of stat data need to be accessible/updatable at every time step
        - Solution: read/write all state data to two float point textures)
      - Update of data needs to be done locally in parallel fashion
        - Solution: most can simulations can be discretized spatially to a local update scheme
  - Cloth simulation algorithm
    - One of earliest physics simulation feat in graphics; 2D manifold embedded in 3D space, elastic material
    - Pseudo-physical model: internal forces (elastic, damping, curvature) and external (gravity, wind, drag)
  - Height-field water simulation and rendering of caustics
    - Represent fluid surface as 2D function h(x,y) or 2D array if discretized (simple, but can't do wave breaks)
    - Caustics: caused by redistribution of photons via refraction through curved surface
      - Ex. Light through water surface on pool bottom, or light through glass on table
      - Assume surface of caustic pattern is diffusive in all directions; generate photo intensity map on surface, then blend that with surface color texture for caustic pattern
    - Two-Pass Photon Map Generation
      - Project surface triangles to bottom via refraction; calculate total photons from surface triangle (area*cos(theta), theta=angle between surface normal and reversed incoming light direction)
      - Add up photon intensity from different projected triangles (turn on blending, set blending to add up fragment colors); photon intensity of projected surface triangle is total photons from triangle divided by area of projected triangle

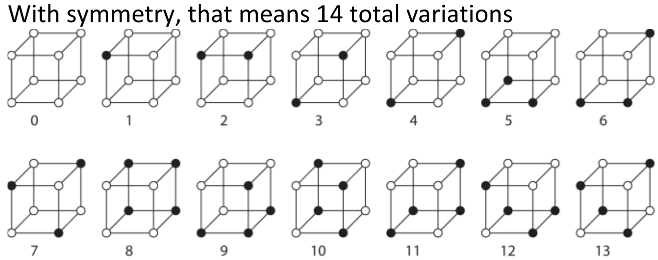
<u>Lecture 10:</u>

- Volume Rendering
  - We've only looked at surfaces in 3D space, not how values over 3D field might be rendered
  - Usually volume rendering deals with how to display a scalar field
    - Function that returns scalar value at particular location: scalar=f(x,y,z)
  - Challenges: lot more data needed to represent volume, managing volumne data is hard (do we store in file?!), how do we display it meaningfully?; here, we try to display internals of object too, not just surface
  - Data: could be sampling of physical process (like medical scan, physical simulation of some type, etc.)
    - Results in a 3D array of scalar sample values
    - One of 3D cells containing sample is a voxel, which represents a scalar value averaging that cell's value
    - Equally spaced 3D array of voxels is often a structured data set; otherwise, it's unstructured data set
  - Rendering
    - Direct volume rendering: use every voxel to produce an image
    - Iso-surface rendering: use implicit equation determines what's rendered

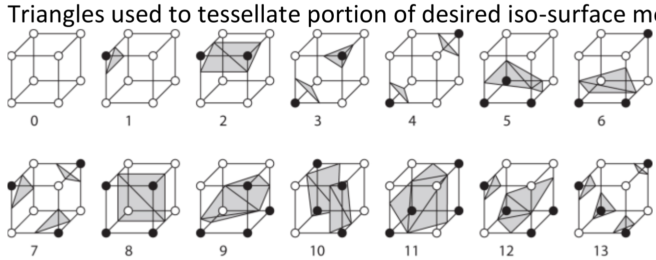
- Rendering iso-surfaces
    - Implicit equation is c=f(x,y,z)
    - Extension of contours to 3D
    - Finding “surface” that exists where the implicit function describing our data equals particular value
    - Scalar values of this data field could represent things like temperature, pressure, density, etc.
    - Want a surface that shows where value is true within data set (ex. where in 3D field is temp=275 deg?)
    - Could project ray from eye/camera through every pixel on display and into data field
        - This requires being able to compute point along the line where value is equal to our target value
            - Ex. Where along the ray is the scalar value equal to value of interest
    - Computationally expensive; no simple way to do this unless we use basic, known, quartic (like sphere)
    - Also, volume data is usually discretized and not continuous; both of these are problems…
    - Marching cubes is the best solution
        - Form a surface mesh representing approximate value in question over discretized scalar field
        - Representation of voxel with corners defined by scalar function f:



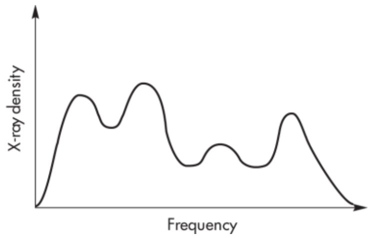
        - Compare vertices of these cells to iso-surface value in question, c
            - Each vertex is colored based on whether value is greater or less than that value
        - With symmetry, that means 14 total variations



        - Specific intersections along each edge can be found by interpolation
        - Triangles used to tessellate portion of desired iso-surface mesh passing through the cell

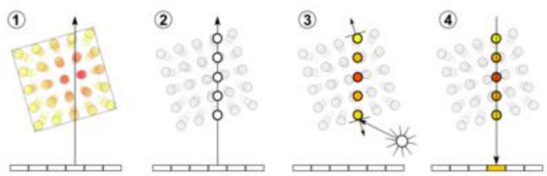


        - Each voxel contributes of eight of these cells
        - When processing, move from one cell to next, row by row, then plane by plane (marching cubes)
        - For each cell processed, the triangles are sent directly to rendering pipeline
            - Reasonable to collect triangles and build more efficient mesh
            - Ambiguities in the resulting mesh are ok, but not enough info in data to always resolve
        - Iso-surfaces make rendering volumes manageable by really reducing amount of data to render
            - Only a fraction of cells get rendered
- Rendering direct methods
    - Iso-surface is useful if you know value specified will show what you’re looking for (also pretty simple)
    - But it only shows a single value, and the rest of the data is ignored
    - To use contributions from all the data, need some mapping of scalar values to color and opacity
        - Otherwise, no idea how each value contributes
    - Color/opacity selection is more art than science, but info on data can help make choices
    - Peaks can tell us which elements are interesting to call out (ex. Bone, soft tissue, etc.)



- Adjust opacity of any color to call out particular structure within the volume
  - Usually user sets values interactive
- This is known as Transfer Function

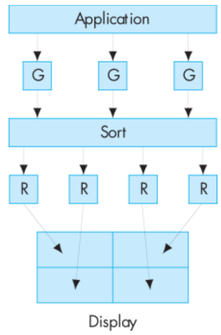
- Splatting: simple method where shape assigned to all voxels
  - Exact shape to use for a splat is undergoing research (often ellipsoids used)
  - Shapes projected/composited back-to-front on viewing plane so transparency/blending is ok
  - Order defined by data grid orientation to view point; spatial data structure (oct-tree) can sort
- Ray casting: similar to implicit method of iso-surface rendering in that we project ray through volume
  - Values along ray are tri-linearly interpolated to color and opacity values



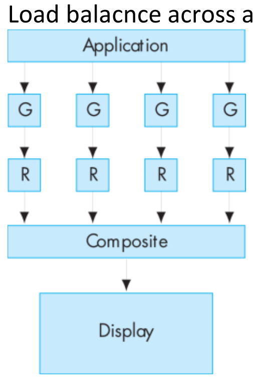
  - Each sample is lit normally using gradient of voxel with respect to light soruce and viewer
  - Finally, samples composited back-to-front order to obtain inal pixel value
  - Very high quality results! GPUs can be used to speed this technique up a lot
- Textures:
  - OpenGL supports 3D textures; each slide/plane of a data volume is stored in texture map
    - Colors/opacities baked onto images
  - Applying 3D texture coordinates to geometry allows arbitrary slices of data to be rendered
  - Sampled values are tri-linearly interpolated from texture data

<u>Lecture 11:</u>

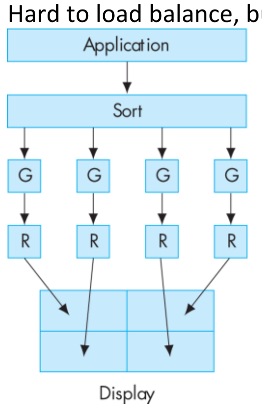
- Parallel Rendering
  - Sometimes one computer isn't enough: not enough pixels OR not enough polygons
  - Not enough pixels
    - Even high-res displays aren't enough (HDTV is 2 megapixels); tile displays together as a "power wall"
    - Tiled displays can reach 150 MP+ (ex. SAGE, a library used to manage these walls developed by UIC)
  - Not enough polygons
    - Parallelizing increases throughput; but how we parallelize depends on problem
    - All graphics parallelizing breaks down into either: sort first, sort middle, or sort last
    - Sort middle: how all GPUs work nowadays
      - Any number of geometry process (G) and/or fragment rasterizers (R)

- 

- Each rasterizer associated with part of display
- Primitives sorted to rasterizing corresponding to projected area of primitive
- Load balances fairly well on GPU; difficult at application level

▪ Sort last: geometry and rasterization handled by single unit
  - Load balacnce across all units
  - 

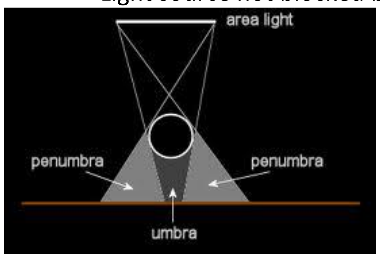
  - Good load balancing for rendering, but we'll have to composite all pieces back together
    - Compositing is a problem…
    - Each rendering system can potentially render to entire display
    - To composite result requires depth information
    - Compositing reads entire color and depth buffer and sends over bus or network
      - All buffers are combined into final image, so fast network is required
      - Load balancing is great, but speed can be an issue
        - Readback, network, depth processing, upload

▪ Sort first: objects ordered to renderer that handles the part of the display it will be projected to
  - Hard to load balance, but compositing is faster
  - 

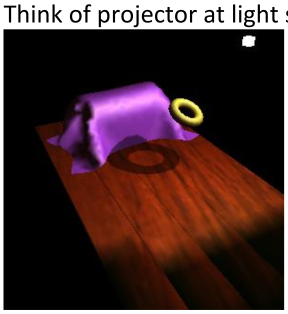
  - Trick is to quickly determine where objects will land onscreen
  - Must estimate how long it will take to render objects in any scheme
    - One way around this is to adjust screen partitions
  - Videos

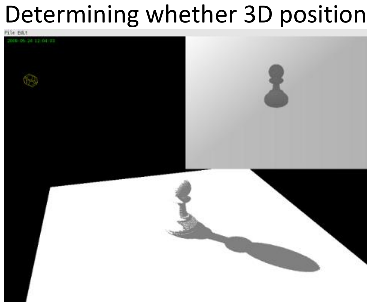
<u>Lecture 12:</u>

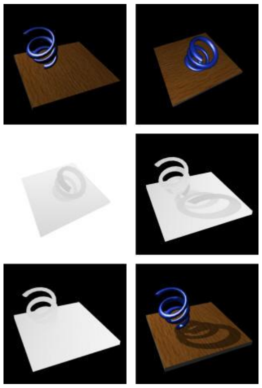
- Shadows
  - Important to the real-world light experience; helps understand shape and spatial relationships between objects
  - Challenging in real-time graphics
    - Algorithms with simple concepts are too slow in real-time
    - Even today, no technique works well in all situations (best approach depends on constraints on situation)

- Def: “area that’s not or partially illuminated due to light being intercepted by opaque object between area/light”
  - Shadow graphics definition: “region of space for which at least one point of light source is occluded”
  - Assumptions: only direct illumination considered (no bouncing light); occluders assumed as opaque
- Shadows in graphics
  - Completely general algorithms extending beyond opaque objects and direct illumination are still very much beyond current hardware capabilities for real-time rendering
  - Accurate shadows are among most important unsolved problems in graphics; we’ll use simple definition
- Shadow components
  - Point p on surface can be one of the following with respect to an area light source
    - Entire light source blocked by scene → p within umbra and in shadow
    - Light source partially blocked by scene → p within penumbra and partially in shadow
    - Light source not blocked by scene → p is lit and not in shadow
  - 

  - Basically, trace a ray from light source l to point p on a surface
    - If intersection between l and p, point p is in shadow
    - Object intersected by ray is the “occlude” (or the “blocker” or “shadow caster”)
    - Object point p belongs to is the receiver
  - To simplify more, we’ll only use point lights to avoid difficult integration needed for area light sources
    - Only using point light sources restricts results to hard shadows only; why?
    - Soft shadows can be achieved using approximation techniques that give appearance of sampling an area light source (ex. Percentage closer filtering); we’re not going to discuss these
- Techniques for hard shadows
  - Shadow mapping (projective & depth)
  - Shadow volumes (not going to discuss)
  - All variations are tradeoff between efficiency and accuracy
  - “Hard” is used because the result is binary, either point is light or it’s not (either in shadow or not)
  - Remember there’s no point light source in the real world - not even a light bulb!
    - Approximation to simplify computation in order to achieve interactive frame rates
- Planar projective shadows
  - Projection of shadow onto a planar (flat) surface only!
  - Lots of problems, not really used ever

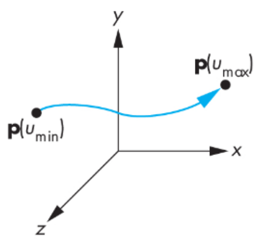
- Projective shadow texture
  - Variation of projective textures that’s not limited to planar receiver surfaces
    - Think of projector at light source casting an image of the shadow onto receivers
    - 
  - Texture buffer is cleared to white
  - Occluders are rendered in black into buffer (a single texture lookup tells us if receiver is in shadow or not)
  - To render into texture, setup camera in light space, NOT camera space
  - Good use for lookAt function to setup appropriate projection matrix
    - Position camera at light source location, in direction of the shadow (light)
  - Results in transformation matrix that transforms vertex from world space to light space
  - Need projection matrix to transform our points into “light clip space”
  - Use orthographic or perspective transformation
    - Common to use orthographic projection for infinite light sources (ex. Sun)

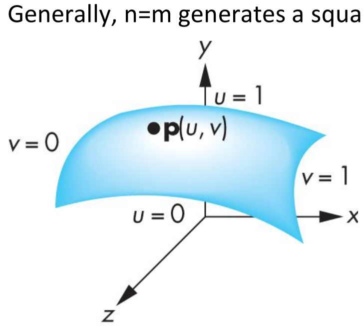
    - Point light source would use perspective transform
    - d is the distance from light source to projection plane
      - $M_{L,V} = [d\ 0\ 0\ 0;\ 0\ d\ 0\ 0;\ 0\ 0\ d\ 0;\ 0\ 0\ -1\ 0]$
  - We're in clip space: everything needs to map to [-1.1], so d is typically 1, which projects to z=-1
  - To keep x, y in same range requires small change to projection matrix with additional scaling values; w, h are size of our shadow texture buffer
    - $M_{L,P} = [w\ 0\ 0\ 0\ ;\ 0\ h\ 0\ 0;\ 0\ 0\ 1\ 0;\ 0\ 0\ -1\ 0]$
  - Not useful for this technique, but for others: maintain depth info rather than projection everything to z=-1
    - Results in familiar perspective projection matrix; you can use Perspective() function
    - Similar to results with a Kinect
  - In vertex shader when rendering receivers, shadow texture's texture coordinates are computed as v^s
    - Need to adjust light clip values [-1,1] to conventional text coordinates [0,1]
      - $M_t = [0.5\ 0\ 0\ 0.5;\ 0\ 0.5\ 0\ 0.5;\ 0\ 0\ 0.5\ 0.5;\ 0\ 0\ 0\ 1]$
      - Entire projection baked into $M_s = M_t * M_{L,P} * M_{L,V}$ and $V^s = M_s*v$
  - Values outside of [0,1] not shadowed (checked for in fragment shader)
  - Issues, both good and bad
    - Must separate occluders and receivers; requires shadow texture per blocker; no self shadowing
    - Can achieve a form of soft shadow by filtering texture map
      - This "light attenuation map" bakes shadows of static lights/occluders/receivers to scene
    - Overall, very simple technique
- Shadow Mapping
  - More general form of projective texture shadows
  - No need to separate occluders from receivers; can handle self-shadowing
  - Again, render from position of light source
    - But render entire scene; every point rendered is implicitly lit (anything else is in shadow)
    - Determining whether 3D position in shadow or not needs checking whether lit in shadow map or not
    - 
  - Simple in theory, hard to do nicely in practice
  - Image spaced based, so artifacts are necessary; shadow map resolution can result in jagged shadow edges
  - We're only interested in depth map created by rendering from light's position
  - Each fragment's position p is transformed into light clip space, like before
  - The x, y components index into the depth map; z value is distance from light source
  - Remember, actual x, y values need to be scaled like projective texture was to [0,1], or p^s
  - Z value is compared to depth value of point under consideration
    - If point in light clip space has depth value > point's value in shadow map, the fragment is hidden or in shadow; otherwise, it's lit and shaded normally
  - Since we've been producing maps via projection and rendering…does this only work for spot (directional) lights?
    - No, typical solution involves rendering 6 fustrums around light source, much like a cube map
      - Spherical maps more efficient in terms of pixel/rendering geometry, but cube maps are simple
      - Some have suggested to compromise with tetrahedrons
  - Bigger issue: sampling shadow map buffer during depth comparison
    - Shadow map has limited precision to represent depth values
    - Shadow map has sampled scene at different resolution than camera
    - Rarely does the point in eye space projected into light space correspond to exact depth map sample
    - Leads to light "leaks" due to z-fighting
    - Occurs mostly when receiver is tilted and discretization of depth values result in incorrect comparison
  - Solution: introduce a "bias", a value which pushes the depth values slightly away from the light source
    - Problem is that there's no rule of thumb for deciding what the bias value should be (need to experiment!)
    - But there's a built-in OpenGL mechanism for applying bias, called glPolygonOffset()
  - From top, left to right: view from camera, view from light, shadow map from light, shadow map from camera, scene depth, final image after compare

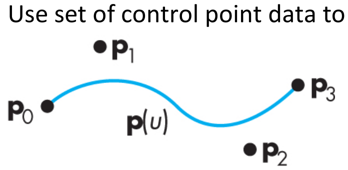
- 
- Of course this assumes we can render to texture; in depth map case, we're rendering depth values to a texture
    - This is done using OpenGL frame buffer object (or FBO)
- In code:
- On setup, create appropriate texture
    - Notice the compare parameters which control how depth comparison will operate
- Next, reate FBO and associate with texture we just defined; finally, render to the FBO

Lecture 13:

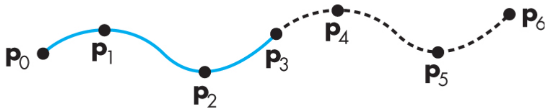
- Curves
    - So far only done flat surfaces
        - Even sphere was approximated by repeatedly sub-dividing into flat sided object (tetrahedron)
        - Convenient because OpenGL designed to render flat objects (AKA triangles) very effieciently
    - New ways to model curves and surfaces
        - Implementation involves flat primitives (lines, triangles) but with description as curves
- Representations
    - Explicit: familiar, but has problems for us
    - Implicit: also familiar, but no analytic form we can use
    - Parametric: less familiar, but most useful for our purposes
- Explicit Representation
    - $y=f(x)$ form with one independent, one dependent variable
    - No guarantee this form exists for a given "curve"
        - ex. Line represented as $y=mx+b$, but can't represent vertical line
        - Likewise, many curves have no explicit form (ex. A loop that looks like gamma)
    - Circle is even more obvious of a problem; can only represent half of circle
        - We'd need to do $y=\sqrt{r^2-x^2}$ AND $y=-\sqrt{r^2-x^2}$, iff $0\le|x|\le r$
    - 3D has similar problems as 2D
        - Curves need 2 equations (dependent variables): $y=f(x)$, $z=g(x)$
        - Surfaces require two independent variables: $z=f(x,y)$
        - Some curves not represented in this form either
            - Lines can't be described if it exists on any plane of constant x, when defined in terms of x
                - Kind of like vertical line problem
            - Spheres can't be described due to same ambiguities as 2D case without constraints
                - $z=f(x,y)$ can generate 0,1, or 2 points on sphere
            - Too many corner cases to be useful for more complex curves and surfaces (patches)
    - Implicit Representation
        - Curves represented using form $f(x,y)=0$
        - Line has usual $ax+by+c=0$ form; circles look like $x^2+y^2-r^2=0$
            - No real representational limitations here!
        - Except implicit form is more useful for testing membership
        - Is point on line, curve, surface, or not? Collision detection!
        - Generally no convenient way to analytically determine x given a y value; this limits usefulness in rendering
    - Same for 3D: we can represent lines, curves, surfaces
        - Lines are $ax+by+cz+d=0$, surfaces are $f(x,y,z)=0$
        - Curves are trickier with intersection of two 3D surfaces: $f(x,y,z)=0$ and $g(x,y,z)=0$
    - While almost any curve/surface we use has implicit representation, extracting points along or on them is difficult!
        - We need those points to draw the curve!

- Parametric Representation
    - Represents each dimensional point on curve with respect to a single independent variable u
        - x=x(y), y=y(u), z=z(u); only use z for 3D representation
        - Varying u allows us to generate points that sweep out the curve
    - Convenient for rendering, but not necessarily fast
    - Surfaces are similar: x=x(u,v), y=y(u,v), z=z(u,v); these can all go in a vector p(u,v)
        - Varying u, v can sweep out a surface
        - Taking cross product of tangent vectors at a point gets us the normal of the surface
            - n = dp/du * dp/dv
    - Curves represented as polynomials in terms of u and surfaces in terms of u, v are convenient to use; we get:
        - Local control of their shape
        - Smoothness and continuity control between curves/shapes
        - Ability to evaluate derivatives
        - Behavioral stability
        - Ease of rendering
    - Example curve
        - p(u)=[x(u), y(u), z(u)]
        - Polynomial representation of degree n is (not dimension): p(u) = sum from k=0 to n (u^k*c_k)
            - Each of n+1 c_k's has independent coefficients, like c_k=[cxk, cyk, czk]
        - 

    - Example surface
        - p(u,v)=[x(u,v), y(u,v), z(u,v)]
        - Polynomial representation of degree n is p(uv) = sigma i=0 to n (sigma j=0 to m (c_ij*u^i*v^j))
        - Again, m+1 and m+1 c_k's have independent coefficients
        - Generally, n=m generates a square shaped surface patch
        - 

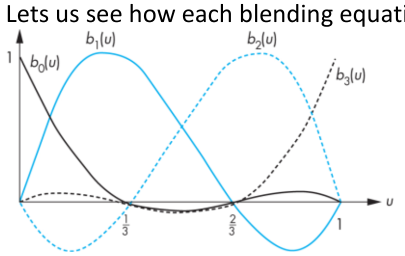
    - Choosing degree: we can use whatever degree we want
        - But high degree results in complex curve that's hard to control
        - Low degree might be too simple to represent what we need
        - Solution: use collection of simpler curves to represent more complex shapes
        - In graphics, most common degree is 3, resulting in cubic polynomials (we'll almost always use this)
        - Degree of 3 keeps shape control local and lets us manage joints between curve segments more easily
            - Degree of 3: p(u)=c_0 + c_1*u + c_2*u^2 + c_3*u^3 = sigma k=0 to 3 (c_k*u^k) = u^T*c
            - c = [c_0, c_1, c_2, c_3]; u=[1, u, u^2, u^3]
        - We have to solve for values of c
            - 12 equations in 12 unknowns for degree of 3
            - X, y, z are independent, so we group the problem into 3 sets of 4 equations with 4 unknowns
            - Use set of control point data to help solve for unknowns, lets us to generate points along curve
            - 

    - Cubic interpolating polynomial

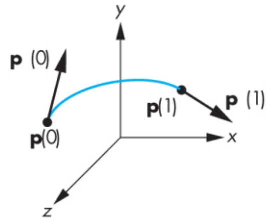
    - Not used often in CG
    - There are other curve types with beneficial properties for control and rendering
    - For now, have four control points in 3D
    - Seek coefficients c such that polynomial p(u)=u^T*c passes through control points
    - 

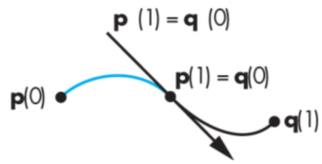
    - Control points defined by p_k = [x_k, y_k, z_k]
    - Also choose values of u at each p to perform interpolation; these are convenient: u=0, 1/3, 2/3, 1
    - This gives four conditions, where
        - p_0=p(0)=c_0
        - p_1 = p(1/3) = c_0 + 1/3*c_1 + (1/3)^2*c_2 + (1/3)^3*c_3
        - and so on, until p_3 = p(1) = c_0+c_1+c_2+c_3
    - Thus we get p=Ac, where
        - p=[p_0, p_1, p_2, p_3]
        - A=[1 0 0 0; 1 1/3 (1/3)^2 (1/3)^3; …; 1 1 1 1]
        - Inverting A gives "interpolating geometry matrix" and the coefficients c = M_I*p
    - Now we can evaluate points along curve
    - To continue our curve, define next segment with p_3, p_4, p_5, and p_6; achieves continuity!
        - Does not ensure the same derivative at point p_3, where curves join
        - May or may not be a problem
    - We mentioned we desired ability to build up complex curves using simpler ones
        - Behavior of how curves join says plenty about how we achieve this!

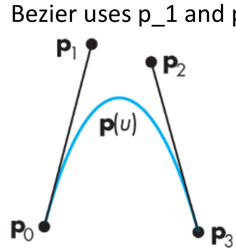
<u>Lecture 14:</u>

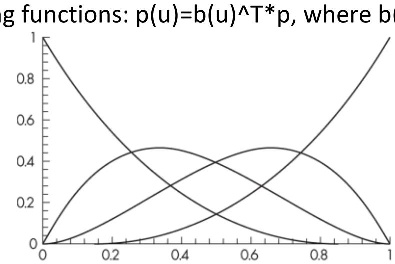
- Curves
- Parametric: Cubic Blending Functions
    - Slightly different way of looking at this interpolation process; it allows us to see what exactly is going on
    - Let's substitute interpolating coefficients into polynomial itself (generalize things a bit)
    - p(u)=u^T*c=u^T*M_I*p, or p(u)=b(u)^T, where b(u)=M_I^T*u
    - b(u)=[b_0(u), b_1(u), b_2(u), b_3(u)]
    - Expressing p(u) in terms of cubic blending polynomials gives us:
        - p(u)=b_0(u)*p_0 + …+ b_3(u)*p_3
        - b_0(u) = -9/2*(u-1/3)(u-2/3)(u-1)
        - b_1(u) = -27/2*u(u-2/3)(u-1)
        - b_2(u) = -27/2*u(u-1/3)(u-1)
        - b_3(u) = 0/2*u(u-1/3)(u-2/3)
        - Lets us see how each blending equation factors into interpolation
        - 

        - Each one isn't particularly smooth (since we're having interpolation pass through each control point)
    - Higher degree polynomials have more pronounced swings
    - Remember there's no way to enforce derivatives at endpoints; makes this form limited
- Parametric: Hermite Form
    - We can form curve/surface using cubic interpolating polynomial, but there are issues
    - Hermit form allows additional control over derivatives at ends of the curve
    - Here, only consider control points p_0 and p_3, which from previous example, has first two conditions
        - p_0 = p(0) = c_0
        - p_3 = p(1) = c_0 + c_1 + c_2 + c_3
    - We get other two conditions if we assume the derivatives at u=0 and u=1 are known
        - P'(u) = c_1 + 2uc_2 + 3u^2c_3
        - p_0' = p'(0) = c1
        - p_3' = p'(1) = c1 + 2c_2 + 3c_3

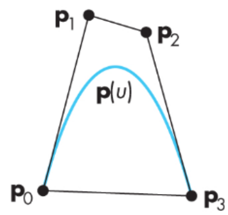
        - matrix form: $q=[p_0, p_1, p_2, p_3]=[1\ 0\ 0\ 0;\ 1\ 1\ 1\ 1;\ 0\ 1\ 0\ 0;\ 0\ 1\ 2\ 3]$
        - 

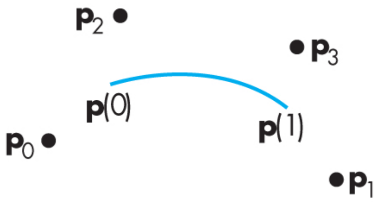
        - Solve for c to get $c=M_H*q$, $M\_[1\ 0\ 0\ 0;\ 0\ 0\ 1\ 0;\ -3\ 3\ -2\ -1;\ 2\ -2\ 1\ 1]$
            - M_H is the Hermite geometry matrix
        - Again, we get resulting polynomials
        - Blending functions can be used in same way as before for cubic interpolating polynomial
        - Hold derivative to be the same across curve segments (at the join) to get continuity
            - 

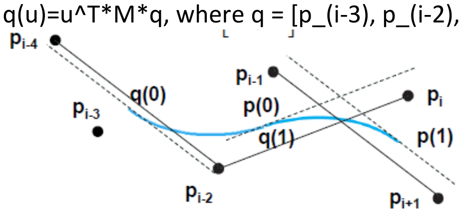
            - Curves connecting at endpoints have C^0 parametric continuity
            - If derivatives also match, we get C^1 parametric continuity
            - If derivatives are proportional to each other, we get G^1 geometric continuity
- Parametric: Bezier Form
    - Can't really compare cubic interpolating polynomial to Hermite
        - Both are cubic in degree; but they don't use same data (control points)
    - Bezier Form: use all four control points of cubic interpolating polynomial to approximate Hermite curve
        - Named after Pierre Bezier, who worked for Renault in France (1960s)
        - Again, use endpoints p_0 and p_3, insisting interpolation passes through these values
            - $p_0 = p(0) = c_0$
            - $p_3 = p(1) = c_0 + c_1 + c_2 + c_3$
        - Bezier uses p_1 and p_2 to approximate tangents at $u=0$ and $u=1$, instead of using them for interpolation
        - 

        - Approximating tangent results in some conditions…
        - Get four equations with four unknowns
            - Solving for c, $c=M_B*p$, where $M_B=[1\ 0\ 0\ 0;\ -3\ -3\ 0\ 0;\ 3\ -6\ 3\ 0;\ -1\ 3\ -3\ 1]$
            - M_B is Bezier geometry matrix
        - Cubic polynomial is then $p(u) = u^T*M_B*p$
            - Exact same manner as cubic interpolating polynomial seen earlier
            - If control points are overlapped, you should see that we still have C^0 continuity at join
            - We do not have C^1 continuity like with Hermite due to different approximations for tangent
    - Blending functions: $p(u)=b(u)^T*p$, where $b(u)=M_B^T*u=[(1-u)^3, 3u(1-u)^2, 3u^2(1-u), u^3]$
        - 

        - Zero values only at end of interval; this ensures smooth interpolation over interval [0,1]
        - Also see that while $0<u<1$
            - Blending functions are also $b(u)<1$; this condition is called a "convex sum"
            - Implies that curve will be contained within convex hull of control points (good interactive design)



- Parametric: Cubic B-Splines
  - Bezier curves only achieve C^0 continuity (mathematically speaking)
  - We can achieve C^1 by matching tangents
  - Relax condition that interpolation must pass through control points to achieve C^2 continuity using cubic B-spline
  - 

  - Previously, we varied u from 0 to 1, the curve spanned over all four control points
    - Instead, consider spanning over only middle two control points
  - Matching conditions at $p(0)$ with $q(1)$ achieves C^2 continuity by solving for M
  - $p(u)=u^T*M*p$, where $p = [p_{i-2}, p_{i-1}, p_i, p_{i+1}]$
  - $q(u)=u^T*M*q$, where $q = [p_{i-3}, p_{i-2}, p_{i-1}, p_i]$
  - 

  - Use symmetric approximations for tangent at joint point (like p_{i-2}, p_{i-1}, and p_i?)to get
    - $p(0) = q(1) = 1/6*(p_{i-2}+4p_{i-1}+p_i) = c_0$
    - $p'(0) = q'(1) = ½(p_i-p_{i-2}) = c_1$
  - Do same conditions at p1), sliding down to next set of control points
    - $p(1) = 1/6*(p_{i-1}+4p_i+p_{i+1}) = c_0+c_1+c_2+c_3$
    - $p'(1) = ½(p_{i+1}-p_{i-1}) = c_1+2c_2+3c_3$
  - Now we have four equations for coefficients of c, so we can solve for M
  - B-spline geometry matrix, $M_S = 1/6*[1\ 4\ 1\ 0; -3\ 0\ 3\ 0; 3\ -6\ 3\ 0; -1\ 3\ -3\ 1]$
  - We now have C^2 continuity at joins, but at a least of 3 times the amount of work!
    - Had to interpolate between each set of control points
  - C^2 continuity not only connects segments, but matches tangents and curvature
    - Very useful properties when modeling real world materials
    - Difficult to use, since curve does not pass through any of control points – not intuitive