3D Game Objects and Levels (Rooms)

Simulating 3D action in 3D world complicates many aspects of a game

- object modelling
  - polygon mesh approximations, multiple coordinate systems (world, intrinsic)
- graphics rendering
  - viewpoints, partial occlusions, perspective, haziness, reflections, stereo vision
- audio
  - Doppler effect, attenuation, echo, stereo/surround sound
- collision detection
  - intersections of polygons, collisions of polyhedra, spheres, polygon meshes
- collision response
  - kinematics and kinetics approaches in 3 dimensions
- geometry and physics and animation
  - combining rotations, avoiding self-intersections, expensive "special effects"
- maintaining acceptable frame rate

Quotes

(Note, the text below was in the context of developing a game by coding in a language like C# or C++, and not using a game engine such as Unity or Unreal)

For one person, it's much more realistic to produce a complete 2D game. The 3D games one person can produce are probably more simplistic than the 2D games because there's so much extra work that goes into 3D.

A game like GTAIV requires a team of hundreds of professional programmers and artists many **years** to produce even though they are working on it at least 5 days a week all year round. Keep that in mind, and realize that even small victories in 3D are victories.

dreamincode.net/forums/topic/291681-best-3d-modelingvideo-game-designing-booksoftware-for-beginners/

Lines and Planes in 3D *vs.* 2D

Whereas in 2D, infinite non-parallel lines must intersect somewhere,
in 3D, they may or may not intersect: they may be *skew lines*

An infinite plane in 3D is commonly represented by a 4-tuple

- X, Y, Z components of a (unit) vector normal to the plane
- signed distance along that normal vector between plane & coordinate system origin

Rotation, translation, scaling, shearing, projection of arbitrarily oriented lines and planes is commonly done using "homogeneous coordinates" and 4x4 matrices

More on 3D geometry in a later lecture …

Polygon Meshes for 3D Shape Representation

Commonplace to represent the surface of a 3D shape using many plane polygons

- even spheres and other rounded objects, approximately
- polys are usually convex, triangles particularly popular, quads less so
- represent *only* the surface: not explicitly the interior of a solid object
  - collision detection might not spot one object existing or moving inside another

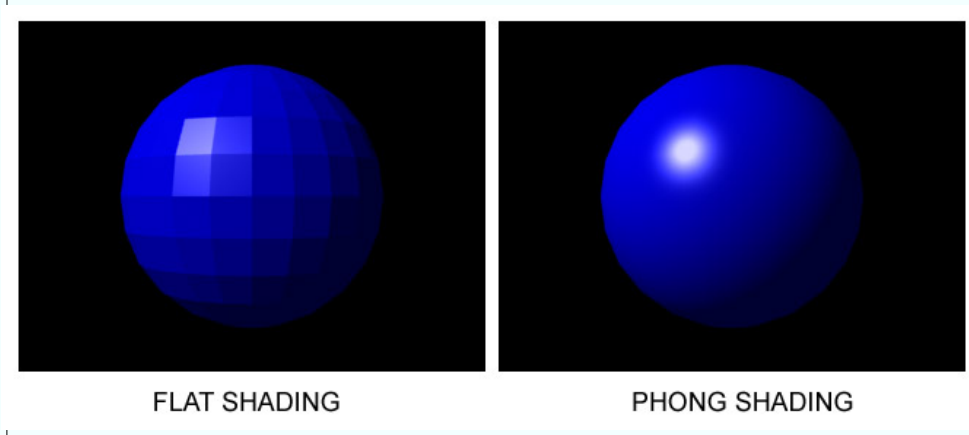
Even with very many polygons, the approximation involved may show up visually

- especially with specular reflections
- (Phong shading, a kind of *interpolation technique*, helps here: pto)

3D models (polygon meshes) can be created using tools such as Blender, 3Ds, . . .

There is a market in such models/meshes. Some showcase models have 1M+ polys

From
https://commons.wikimedia.org/wiki/File:Phong-shading-sample.jpg



Several methods for drawing 3D scenes

One method is the **Painter's Algorithm**

- paint furthest polygons first, overpaint with progressively nearer ones
  - but must you sort the polygons by depth ? and each frame, if objects move?
  - what if several polygons overlap each other cyclically?
- does not help with collision detection, that must be separately done
- partially transparent polygons can alter the colour of what is seen behind them

Another method involves **Binary Space Partitioning Trees** ("BSP" Trees) ("BSPTs")

Other methods use **"Z buffering"**

Two Z-buffering methods for drawing 3D scenes

On a screen, each pixel naturally has an x & y (or h and v). Add a z.

One technique that uses a "Z-buffer"

- for each pixel on screen, trace a line into the simulated world
- when a poly is found, compare its distance with the closest known (initially ∞)
- if closer, update closest distance and paint that pixel (possibly overpainting)
- again does not help with collision detection, that must be separately done

Another technique that uses a Z-buffer:

- for each polygon in the simulated world, calculate the pixels to be drawn
- at each pixel, compare the distance of the part of that polygon with the distance of the part of any other polygon already drawn at that pixel (initially ∞)
- if it is closer, draw the pixel and update the distance

Neither technique is good with partial transparency

BSP (Binary Space Partitioning) Trees (BSPTs)

BSPTs are especially well suited to organising the polys representing a rigid object

- **-ve:** Building the tree initially is a fairly expensive operation, **O**(N.logN)
- **+ve**: The cost is amortised over many frames with cheaper rendering and search
- **+ve:** Multiple BSPTs can be merged efficiently, moving objects can be combined

The basic idea is to partition all the polys in a scene into

- a particular *pivot* poly and any others coplanar with it
- those that lie "to front" of the pivot poly (there may be none such)
- those that lie "to rear" of it (again, there may be none such)
  - polys that intersect pivot's plane are cut into two, one behind, one in front
  - if not convex, or if you insist on triangles, then maybe more than two

Polys to front are recursively partitioned using one as pivot; likewise, polys to rear.

Painter's Algorithm using BSPTs

To paint a scene represented by a BSP Tree, from *any* pov (camera position)

- are you looking at its chosen poly from "front", from "rear", or edge-on?
  - if "from front",
    - (recursively) paint the scene "to rear" of it
    - paint its front face (different textures may apply to front & back)
    - paint other polys coplanar with it, their fronts or backs as appropriate
    - paint the scene "to front" of it
  - if "from rear",
    - paint the "to front" partition,
    - then the pivot's back, and others coplanar,
    - and then the "to rear" partition
  - if edge-on,
    - ignore it and those coplanar; you don't see them as they are edge-on
    - paint "to front" and "to rear" partitions in either order, they won't overlap

Combining BSPTs

The scene to be painted probably consists of many objects each with their own BSPTs. BSPTs may be efficiently merged, allowing BSPT for entire scene to be assembled.

To assist merging, each BSPT should be associated with a 3D region,

- as an OBB (Oriented Bounding Box)
  - OBBs are easily computed from their pivot poly(s) and the OBBs of their to-front and to-rear BSPTs (although they quickly grow very large if done this way)
- as an AABB
  - even easier to compute
  - may be excessively large for thin non-axis aligned BSPs
- as a convex hull
  - much less easy to compute, but most economical in spatial terms

Combining BSPTs (2)

When combining $BSPT_1$ with $BSPT_2$,

- if $BSPT_2$'s region is wholly to-front of $BSPT_1$'s pivot poly(s),
  - recursively combine $BSPT_2$ with $BSPT_1$'s to-front BSPT,
  - make new BSPT with same polys, same to-rear, newly built to-front
- likewise if wholly to-rear, (but swap wording as appropriate),
- if plane of $BSPT_1$'s pivot poly(s) intersects $BSPT_2$'s region,
  - either trivial, if $BSPT_1$'s pivot poly(s) are coplanar with $BSPT_2$'s
    - make BSP with both sets of pivots, merged front parts, merged back parts
    - especially trivial if some front or back parts are empty
  - *or not trivial . . .*

When $BSPT_1$'s pivot plane nontrivially intersects $BSPT_2$'s 3D Region

- Split $BSPT_2$'s pivot poly(s) into (up to 2) parts (to front, to rear)

- Keep its to-front part if its region is wholly to-front or wholly to-rear of the plane, otherwise split it (recursively, by this method) into a new to-front and a new to-rear
- Similarly with its to-rear part
- Keep separate any polys that happen to be coplanar with the intersecting plane

Are there any to-front parts of $BSPT_2$'s pivot poly(s)?

- if not, combine all the front-of-plane fragments into one (as prev.); otherwise
  - use them as new pivot polys of a new BSPT
  - make its to-front from front-of-plane fragments of $BSPT_2$'s to-front
  - similarly, make its to-rear from rear-of-plane fragments of $BSPT_2$'s to-rear

Likewise with any to-rear parts

Eventually, combine the new BSPTs (and any coplanar polys) with $BSPT_1$.

Other properties of BSP Trees

In order to speed up search for collisions (understood as intersecting surface polygons), trees should **not** necessarily be balanced.

- It is more desirable to have trees where there are large regions with few polygons, and small regions with many polygons
- This allows the commonest case (no-collision) to be detected more quickly, there is a greater probability that some object is in the larger region than in the smaller one

Trees carve up 3D space into progressively smaller regions.

- Multi-resolution spatial reasoning (and rendering) is possible;
- With well-chosen pivot planes (such as those on objects' convex hulls) good approximations to their overall shape can be got by limiting the depth of tree that is considered.

See *www.bowdoin.edu/~ltoma/teaching/cs350/fall13/Material/bsp_tutorial.pdf*

Other 3D spatial representations: Voxels

Voxels – divide 3D space into uniform cubes, like pixels divide 2D into squares

This way of representation has many drawbacks

- voxels will be very numerous especially for game levels set outdoors;
- need inside/outside distinction, containing object information
- for exterior voxels, colour and/or texture info needed for rendering;
- a lot of data shuffling will occur when objects move;
- granular for both motion purposes and for visuals (cf. anti-aliasing in 2D)

Other 3D spatial representations: octrees

Octrees – like quadtrees, each node either is a leaf or has eight daughters

coordinate ranges should ideally be power of two in all 3 directions
index daughter node by most significant bits of 3 coordinates, then leftshift them

- eg with 32x32x32 cube, a coordinate at <12,22,17> will be in
  <01100, 10110,10001> (ie division 011, decimal 3) then
  <01100, 10110, 10001> (its subdivision 100, decimal 4) then
  <01100, 10110, 10001> (its subdivision 110, decimal 6) … and so on

empty regions may be represented by vastly fewer nodes than voxels
(parts of) objects listed as occupying octree node must have colour/texture
issue: how does octree structure change as objects move?

- will existing cells be split as multiple objects are found to be within?
- will 7/8ths empty octets be simplified?

Other 3D spatial representations: portals

Portals – doors, windows, turns along an outdoor pathway, level advancement

- 3D world is divided into "rooms" with portals connecting adjacent rooms
- A camera in a room sees inside that room; into adjacent ones through its portals
- Portals (e.g. doors) may be open or shut
  - see through open ones only
  - portal itself occupies both rooms
- Further portals may be seen through, and still further ones through them
- Much of what is in an adjacent room may be culled, not painted at all

Issues:

- what room is camera in?
  - can camera move without passing through portals?
  - can viewer go through walls, or teleport? then hunt the camera!
- potentially very many portals
  - consider an outdoor scene with skyscrapers: is each window a portal?

Other 3D spatial representations: Potentially Visible Sets

Like with portals, space is divided into rooms (or "nodes")
Each room holds a list of others that might be seen from any point within it

- Conceptually simple
- For rendering, offers rapid elimination of many polygons that cannot be seen

Issues:

- Inherently conservative: a room **B** must always be in PVS of room **A**
  - if a door might be opened, regardless of whether it is open now
  - if **B** may be seen from *somewhere* in **A**, regardless of where camera is now
- Changing geometry (if doors or walls might get destroyed) is hard to handle
- Automatic generation of PVS is not easy
  - must every possible camera position be considered? if not …
  - geometry of room and/or camera position probably must be constrained