Development of games

Dr. A.V.Gavrilov
Kyung Hee University

Lecture 4

Foundations of Graphics for Games

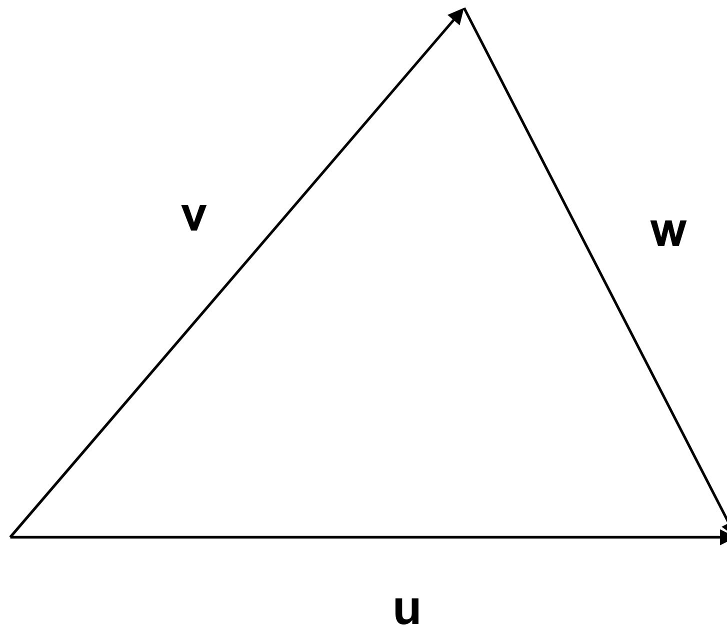
Basically based on book *James M. Van Verth, Lars M. Bishop. Essential mathematics for games & interactive applications. A Programmer’s guide.* Elsiever, 2005.

Vectors

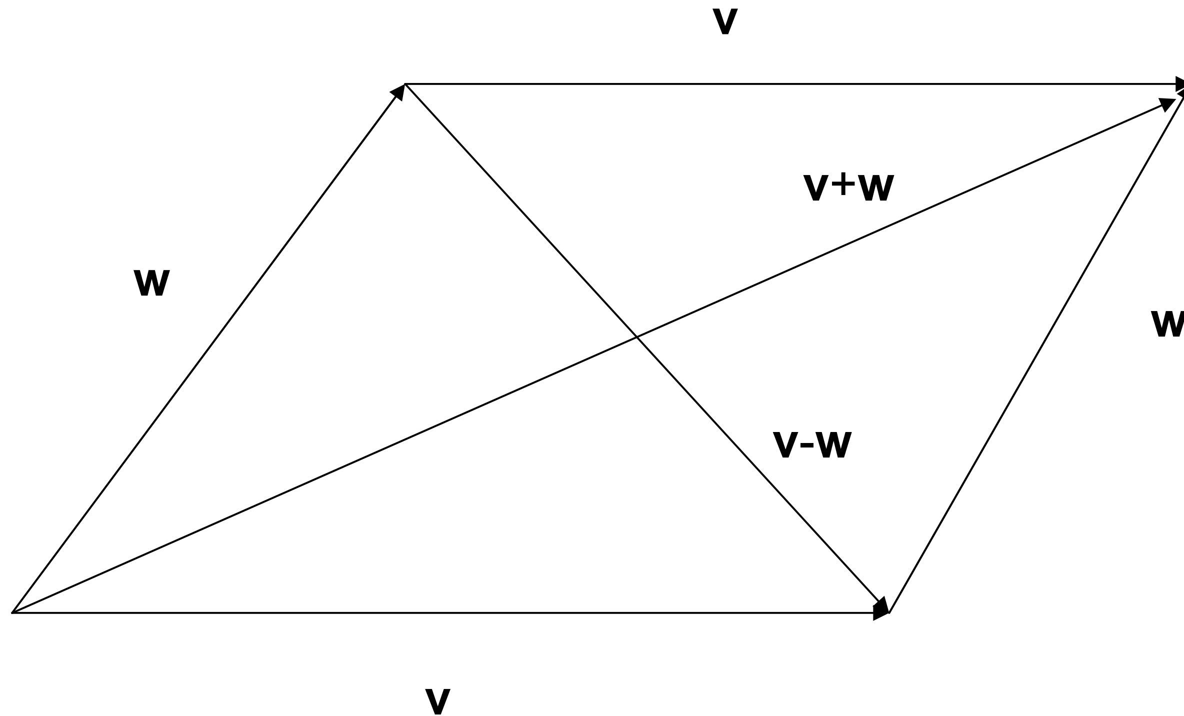
- Vector **v** is an entity with magnitude (length) and direction
- Vector with magnitude 1 is normalized vector
- Not that does not have a location
- Two vectors with same magnitude and direction are equal, no matter where draw on the page (screen)

Vector addition

- **u = v+w**



Vector addition and subtruction



Real Vector Space

- Vector space allows to represent vectors symbolically
- Real vector space is the set of all ordered pairs of real numbers

- 2-dimensinal $\Re^2 = \{(x, y) \mid x, y \in \Re\}$
- 3-dimensinal $\Re^3 = \{(x, y, z) \mid x, y, z \in \Re\}$

...

Standard basis for a vector space

$$\mathbf{e}_0 = (1, 0, ..., 0)$$
$$\mathbf{e}_1 = (0, 1, ..., 0)$$
$$...$$
$$\mathbf{e}_{n-1} = (0, 0, ..., 1)$$

Main property of basis β is that for every vector **v** in V there is a unique linear combination of the vectors in β that equal **v**.

$$\mathbf{v} = a_0\mathbf{b}_0 + a_1\mathbf{b}_1 + ... + a_{n-1}\mathbf{b}_{n-1}$$

or for 3D vector space

$$\mathbf{v} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

We can think of *x,y* and *z* as the amounts we move in the **I, j** and **k** directions.

Inner product

- Set of functions $\langle \mathbf{v},\mathbf{w} \rangle$ returning a real scalar with following properties:

  $\langle \mathbf{v},\mathbf{w} \rangle = \langle \mathbf{w},\mathbf{v} \rangle$

  $\langle \mathbf{u}+\mathbf{v},\mathbf{w} \rangle = \langle \mathbf{u},\mathbf{w} \rangle + \langle \mathbf{v},\mathbf{w} \rangle$

  $a\langle \mathbf{v},\mathbf{w} \rangle = \langle a\mathbf{v},\mathbf{w} \rangle$

  $\langle \mathbf{v},\mathbf{v} \rangle >= 0$

  $\langle \mathbf{v},\mathbf{v} \rangle = 0$ iff $\mathbf{v} = 0$

Dot product

$\mathbf{v} \cdot \mathbf{w} = ||\mathbf{v}||||\mathbf{w}|| \cos\theta$

$\mathbf{v} \cdot \mathbf{w} = u_x w_x + u_y w_y + u_z w_z$

If $\mathbf{v} \cdot \mathbf{w} = 0$ then vectors **v** and **w** are orthogonal

Let in game **v** is a view vector and **t** is vector of any object.

If $\mathbf{v} \cdot \mathbf{w} < 0$, then the object is behind us and therefore is invisible for AI

Cross product

- Aim is determination new vector orthogonal to both determined vectors
- Also it is known as *vector product*

$$\|\mathbf{v} \times \mathbf{w}\| = \|\mathbf{v}\|\|\mathbf{w}\| \sin\theta$$

and $\mathbf{v} \times \mathbf{w} = -(\mathbf{w} \times \mathbf{v})$

$$\mathbf{v} \times \mathbf{w} = (v_y w_z - w_y v_z,\ y_z w_x - w_z v_x,\ v_x w_y - w_x v_y)$$

Triple products

- Vector triple product

$$\mathbf{u} \times (\mathbf{v} \times \mathbf{w})$$

- Scalar triple product

$$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w})$$

- If $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) > 0$ then the shortest rotation from v to w is in counterclockwise direction, otherwise – in clockwise direction

Affine space

- An affine space consists of a set of points W and a vector space V
- Relation between the points and vectors:
  - For every pair of points P and Q in W exist a unique vector **v** in V such that

$$\mathbf{v} = Q - P$$

  - For every point P in W and every vector **v** in V exist a unique point Q such that

$$Q = P + \mathbf{v}$$

Representation of any point in W is $P = O + \mathbf{v}$, where

O is fixed point in W named as *origin*

Combination of the origin O and basis vectors is known as a *coordinate frame*

If we work with standard origin (0,0,..,0) a standard basis then we work with *Cartesian frame*

Polar and spherical coordinates

- Relation between polar and Cartesian coordinates

$$x = r \cos \theta,$$
$$y = r \sin \theta$$

- Relation between spherical and Cartesian coordinates

$$X = \rho \sin \varphi \cos \theta,$$
$$Y = \rho \sin \varphi \sin \theta,$$
$$Z = \rho \cos \varphi,$$

where φ is angle between z-axis and projection of **v** on yz plane, θ is angle between x-axis and projection of **v** on xy plane

Triangle

- Determined by 3 vectors

$$\mathbf{v}_0 = P_1 - P_2$$

$$\mathbf{v}_1 = P_2 – P_1$$

$$\mathbf{v}_3 = P_0 – P_2$$

Linear transformations

- Only operations possible in linear transformations are multiplication by a constant and addition
- Linear transformation T is a mapping between two vector spaces V and W, where for all **v** in V and all scalars a:
  - $T(\mathbf{v}_0+\mathbf{v}_1) = T(\mathbf{v}_0)+T(\mathbf{v}_1)$ for all $\mathbf{v}_0,\mathbf{v}_1$ in V,
  - $T(a\mathbf{v}) = aT(\mathbf{v})$ for all **v** in V

Matrices

- It is 2-dimensional array of value (n x m)
- Elements, rows and columns
- Diagonal and trace
- Square, zero, diagonal and triangular matrix

Simple operations

- Addition

$$\mathbf{S} = \mathbf{A} + \mathbf{B}$$

$$s_{i,j} = a_{i,j} + b_{i,j}$$

- Scalar multiplication

$$\mathbf{P} = s\mathbf{A}$$

- Transpose

$$\mathbf{A}^T \quad \begin{bmatrix} 2,-1 \\ 0,2 \\ 6,3 \end{bmatrix} = \begin{bmatrix} 2,0,6 \\ -1,2,3 \end{bmatrix}$$

Matrices (cont.)

- Vector may be represented by matrix with one column (or row)
- Block matrix or matrix with submatrices

$$\begin{bmatrix} 2,3,0 \\ -3,2,0 \\ 0,0,1 \end{bmatrix} \text{may be presented as}$$

$$\begin{bmatrix} \mathbf{A},0 \\ \mathrm{o}^{T},1 \end{bmatrix} \text{where A} = \begin{bmatrix} 2,3 \\ -3,2 \end{bmatrix}$$

Matrix product

C = AB

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

It is used for representation of linear transformation of vector

B = Ax,

where x is n-dimensional vector and b is m-dimensional vector (result of transformation)

Affine transformations

- For points P and constants *a* in affine space A

$T(a_0 P_0 + \ldots + a_{n-1} P_{n-1}) = a_0 T(P_0) + \ldots + a_{n-1} T P_{n-1}$

where $a_0 + \ldots + a_{n-1} = 1$

Affine transformations are useful for games because provide remain of collinearity and coplanarity during transformations

Transformation is $V_{new} = V_{old} \times M$

Affine transformations (cont.)

- Translation
  - T(P) = P + **t**
  - Simulation of moving of object in space, size and shape of object are not changed
- Matrix for translation of vector

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

Affine transformations (cont.)

Rotation

- Around x-axis

  y' = ycosθ - zsinθ

  z' = ycosθ + zcosθ

$$\mathbf{R_x} = \begin{bmatrix} 1,0,0 \\ 0,\cos\theta,-\sin\theta \\ 0,\sin\theta,\cos\theta \end{bmatrix}$$

- Around y-axis

  X' = zsinθ – xcosθ

  Z' = zcosθ – xsinθ

$$\mathbf{R}_y = \begin{bmatrix} \cos\theta,0,\sin\theta \\ 0,1,0 \\ -\sin\theta,0,\cos\theta \end{bmatrix}$$

- Around z-axis

  X' = xcosθ – ysinθ

  Y' = xsinθ + ycosθ

$$\mathbf{R}_z = \begin{bmatrix} \cos\theta,-\sin\theta,0 \\ \sin\theta,\cos\theta,0 \\ 0,0,1 \end{bmatrix}$$

Affine transformations (cont.)

Rotation

$$\mathbf{R_x R_y R_z} = \begin{bmatrix} CyCz, -CySz, Sy \\ SxSyCz + CxSz, -SxSySz + CxCz, -SxCy \\ -CxSyCz + SxSz, CxSySz + SxCz, CxCy \end{bmatrix}$$

Where

Sx = $\text{Sin}\theta_x$ and so on

Affine transformations (cont.)

Scaling

- C_s – center of scaling
- Let C_s=0 and **y**=0
- Then matrix is
- If a=b=c then it is uniform scaling

$$\mathbf{S}_{\mathbf{abc}} = \begin{bmatrix} a,0,0,0 \\ 0,b,0,0 \\ 0,0,c,0 \\ 0,0,0,1 \end{bmatrix}$$

Affine transformations (cont.)

- Reflection
- Shear
  - Shear by x-axis

$$\mathbf{H_x} = \begin{bmatrix} 1,0,0 \\ a,1,0 \\ b,0,1 \end{bmatrix}$$

Combination of transformations

- $V_{new} = V_{old} \times (M_1 \times M_2 \times \ldots)$

- Lot of transformations in games
- So it is necessary hardware support

- graphics processors

3D models

- **3D model** is a mathematical representation of a three-dimensional object
- It can be displayed as a two-dimensional image through a process called *3D rendering* or used in a computer simulation of physical phenomena
- 3D models are most often created with special software applications called 3D modelers. Being a collection of data (points and other information)
- 3D models can be created by hand or algorithmically (procedural modeling) and usually are storing in separate files

3D models (cont.)

- A 3D model by itself is not visual. It can be rendered as a simple wireframe at varying levels of detail, or shaded in a variety of ways
- Many 3D models, however, are covered in a covering called a texture (the process of aligning the texture to coordinates on the 3D model is called texture mapping).
- A texture is nothing more than a graphic image, but gives the model more detail and makes it look more realistic. A 3D model of a person, for example, looks more realistic with a texture of skin and clothes, than a simple monochromatic model or wireframe of the same model

3D model (cont.)

- Other effects, beyond texturing, can be done to 3D models to add to their realism. For example, the surface normals can be tweaked to effect how they are lit, certain surfaces can have bump mapping applied and any other number of 3D rendering tricks can be applied.
- 3D models are often animated for some uses. They can be animated from within the 3D modeler that created them or externally. Often extra data is added to the model to make it easier to animate. For example, some 3D models of humans and animals have entire bone systems so they will look realistic when they move and can be manipulated via joints and bones

Primitives

- Model is based on primitives
- *Primitives* are basic shapes
- Most 3d packages have same primitives:
  - Polygon, Sphere, Cube, Cylinder, Plane
- Most used primitive is polygon

Problems with Polygons

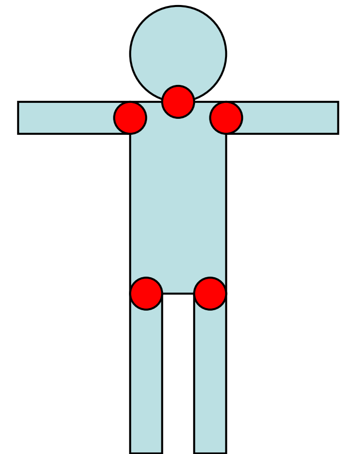
- Interaction is a problem
  - Dragging points around is time consuming
  - Maintaining things like smoothness is difficult
- They are inherently an approximation
  - Things like silhouettes can never be perfect without very large numbers of polygons, and corresponding expense
- Low level representation
  - Eg: Hard to increase, or decrease, the resolution
  - Hard to extract information like curvature

More Object Representations

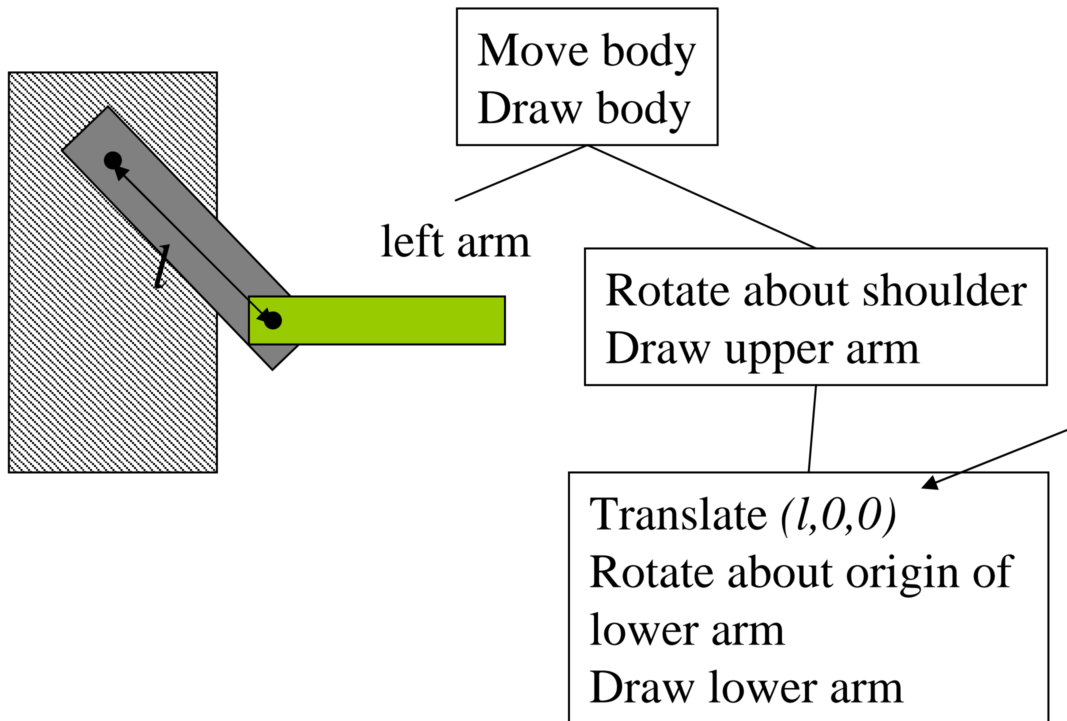
- Hierarchical modeling
- Instancing and Parametric Instancing
- Constructive Solid Geometry
- Sweep Objects
- Octrees
- Blobs and Metaballs and other such things

Hierarchical Modeling

- Hierarchical model: Group of meshes related by a tree (or graph) structure
  - Properties of children are derived from their parents
  - Most useful for animating polygonal meshes
- Consider a walking (humanoid, classic) robot:
  - How would you move the robot around?
  - Does the entire robot move in the same way?
  - Does the position of one part of the robot depend on other parts?



Hierarchical Model Example



Important Point:

- Every node has its own local coordinate system.
- This makes specifying transformations much easier.
- What are we assuming about the “upper arm” coordinate system?

Hierarchical Details

- Generally represented as a tree, with transformations and instances at any node
  - Can use a general graph, but resolving inheritance conflicts is a problem
- Rendered by traversing the tree, applying the transformations, and rendering the instances
- Particularly useful for animation
  - Human is a hierarchy of body, head, upper arm, lower arm, etc…
  - Animate by changing the transformations at the nodes
- Other things can be inherited (colors, surface properties)

Instancing

- Sometimes you need many copies of the "same" object
  - Like chairs in a room
- Define one chair, the base or the prototype
- Create many *instances* (copies) of it, and apply a different transformation to each

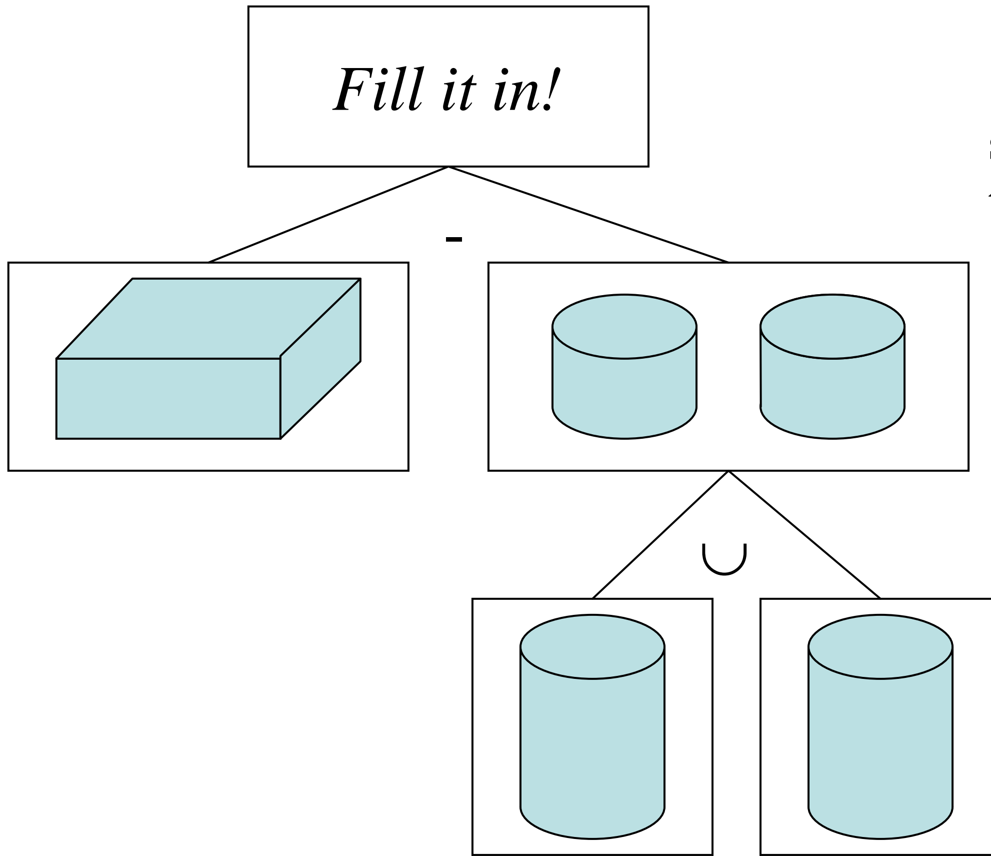
Parametric Instancing

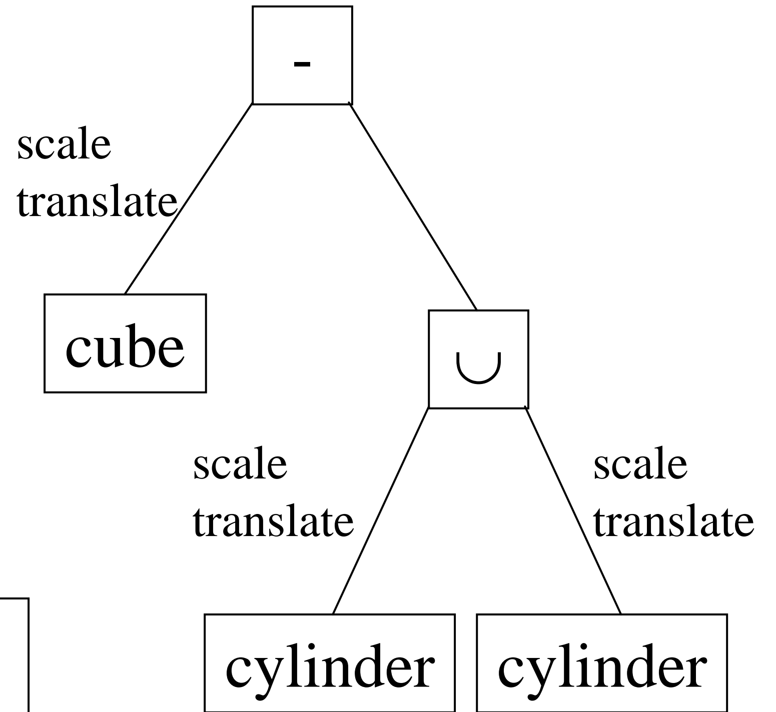
- Many things, called primitives, are conveniently described by a label and a few parameters
  - Cylinder: Radius, length, does it have end-caps, …
  - Bolts: length, diameter, thread pitch, …
- This is a modeling format:
  - Provide software that knows how to draw the object given the parameters, or knows how to produce a polygonal mesh
  - How you manage the model depends on the rendering style
  - Can be an exact representation

Constructive Solid Geometry (CSG)

- Based on a tree structure, like hierarchical modeling, but now:
  - The internal nodes are set operations: union, intersection or difference (sometimes complement)
  - The edges of the tree have transformations associated with them
  - The leaves contain only geometry
- Allows complex shapes with only a few primitives
  - Common primitives are cylinders, cubes, etc, or quadric surfaces
- Motivated by computer aided design and manufacture
  - *Difference* is like drilling or milling
  - A common format in CAD products

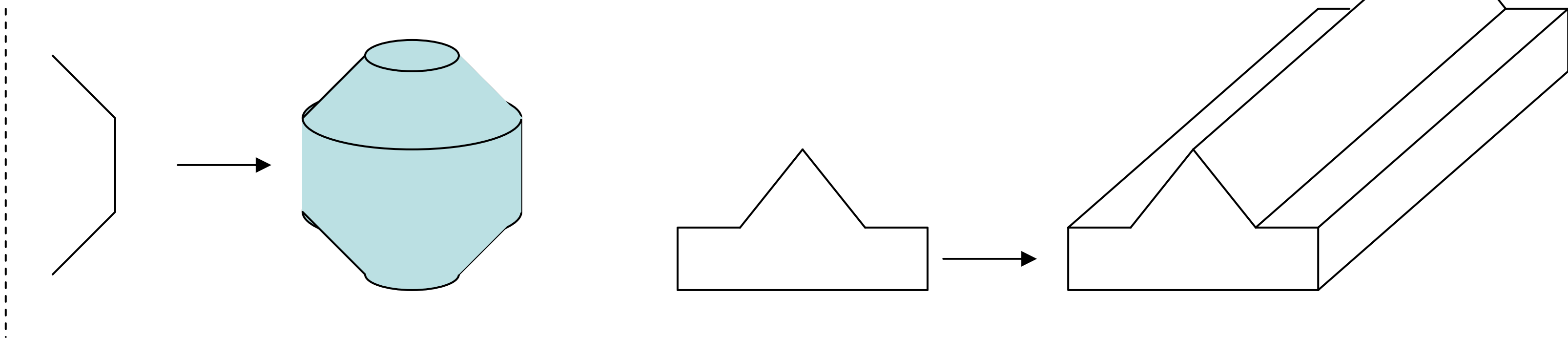
CSG Example





Sweep Objects

- Define a polygon by its edges
- Sweep it along a path
- The path taken by the edges form a surface - the sweep surface
- Special cases
  - Surface of revolution: Rotate edges about an axis
  - Extrusion: Sweep along a straight line



Rendering Sweeps

- Convert to polygons
  - Break path into short segments
  - Create a copy of the sweep polygon at each segment
  - Join the corresponding vertices between the polygons
  - May need things like end-caps on surfaces of revolution and extrusions
- Normals come from sweep polygon and path orientation
- Sweep polygon defines one texture parameter, sweep path defines the other

General Sweeps

- The path maybe any curve
- The polygon that is swept may be transformed as it is moved along the path
  - Scale, rotate with respect to path orientation, …
- One common way to specify is:
  - Give a poly-line (sequence of line segments) as the path
  - Give a poly-line as the shape to sweep
  - Give a transformation to apply at the vertex of each path segment
- Difficult to avoid self-intersection

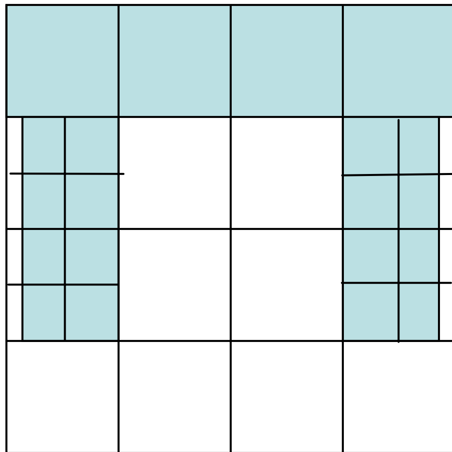
Spatial Enumeration

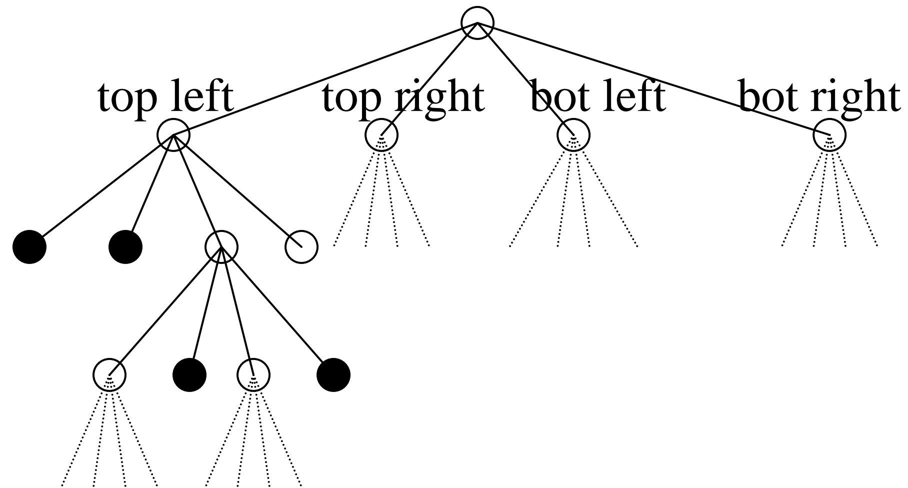
- Basic idea: Describe something by the space it occupies
  - For example, break the volume of interest into lots of tiny cubes, and say which cubes are inside the object
  - Works well for things like medical data
    - The process itself, like MRI or CAT scans, enumerates the volume
    - Data is associated with each voxel (volume element)
- Problem to overcome:
  - For anything other than small volumes or low resolutions, the number of voxels explodes
  - Note that the number of voxels grows with the *cube* of linear dimension

Octrees (and Quadtrees)

- Build a tree where successive levels represent better resolution (smaller voxels)
- Large uniform spaces result in shallow trees
- Quadtree is for 2D (four children for each node)
- Octree is for 3D (eight children for each node)

Quadtree Example





Octree principle is the same, but there are 8 children

Rendering Octrees

- Volume rendering renders octrees and associated data directly
  - A special area of graphics, visualization, not covered in this class
- Can convert to polygons by a few methods:
  - Just take faces of voxels that are on the boundary
  - Find iso-surfaces within the volume and render those
  - Typically do some interpolation (smoothing) to get rid of the artifacts from the voxelization
- Typically render with colors that indicate something about the data, but other methods exist

Spatial Data Structures

- Octrees are an example of a spatial data structure
  - A data structure specifically designed for storing information of a spatial nature
    - E.g. Storing the location of fire hydrants in a city
- In graphics, octrees are frequently used to store information about where polygons, or other primitives, are located in a scene
  - Speeds up many computations by making it fast to determine when something is relevant or not
  - Just like BSP trees speed up visibility
- Other spatial data structures include BSP trees, KD-Trees, Interval trees, …

Implicit Functions

- Some surfaces can be represented as the vanishing points of functions (defined over 3D space)
  - Places where a function *f(x,y,z)=0*
- Some objects are easy represent this way
  - Spheres, ellipses, and similar
  - More generally, quadratic surfaces:

$$ax^2 + bx + cy^2 + dy + ez^2 + fz + g = 0$$

  - Shapes depends on all the parameters *a,b,c,d,e,f,g*

Blobs and Metaballs

- Define the location of some points
- For each point, define a function on the distance to a given point, *(x,y,z)*
- Sum these functions up, and use them as an implicit function
- Question: If I have two special points, in 2D, and my function is just the distance, what shape results?
- More generally, use Gaussian functions of distance, or other forms
  - Various results are called blobs or metaballs

Example with Blobs

Rendered with POVray. Not everything is a blob, but the characters are.

Blob Math

- Implicit equation: $$f(x,y,z) = \sum_{i=1}^{n_{blobs}} w_i g_i(x,y,z) = 0$$
- The *wi* are weights – just numbers
- The *gi* are functions, one common choice is:

$$g_i(\mathbf{x}) = e^{\frac{-(\mathbf{x}-c_i)^2}{\sigma_i}}$$

  – *ci* and σ*i* are parameters

Rendering Implicit Surfaces

- Some methods can render then directly
  - Raytracing - find intersections with Newton's method
- For polygonal renderer, must convert to polygons
- Advantages:
  - Good for organic looking shapes e.g. human body
  - Reasonable interfaces for design
- Disadvantages:
  - Difficult to render and control when animating
  - Being replaced with subdivision surfaces, it appears

Main class of game

```
class IvGame
{
public:
    // Create needs to be implemented in the derived Game
       class
    static bool Create();

    static void Destroy();

    virtual bool Initialize( int argc, char* argv[] );

    void Update();
    void Display();
    inline void Quit()          { mQuit = true; }
    inline bool IsRunning()     { return !mQuit; }

    static IvGame*  mGame;          // global pointer

    IvClock*        mClock;         // main clock
    IvDisplay*      mDisplay;       // window management
    IvEventHandler* mEventHandler;  // event handling

protected:
    // constructor/destructor
    IvGame();
    virtual ~IvGame();

    bool ParseCommandLine( int argc, char* argv[] );
    bool SetupSubsystems();

    virtual void UpdateObjects( float dt ) = 0;
    virtual void Render() = 0;

    bool mQuit;
    bool mPaused;

private:
    // copy operations
    IvGame(const IvGame& other);
    IvGame& operator=(const IvGame& other);
};
```

Fragment of game core routines (1)

```
// @ IvGame::Create()
//-------------------------------------------------
// Static constructor
//-------------------------------------------------
bool
IvGame::Create()
{
   IvGame::mGame = new Game();
   return ( IvGame::mGame != 0 );

}   // End of IvGame::Create()
```

```
/// @ Game::Initialize()
//-------------------------------------------------------------------------
// Set up internal subsystems
//-------------------------------------------------------------------------
bool
Game::Initialize( int argc, char* argv[] )
{
    // Set up base class
    if ( !IvGame::Initialize( argc, argv ) )
       return false;

    mPlayer = new Player();
    if (!mPlayer)
       return false;

    // Set some lights
    ::IvSetDefaultLighting();

    return true;
}   // End of Game::Initialize()
```

Fragment of game core routines (2)

```
// @ Game::Update()
//-----------------------------------------------
// Main update loop
//-----------------------------------------------
void
Game::UpdateObjects( float dt )
{
   // update player
   mPlayer->Update( dt );

}   // End of Game::Update()
```

```
// @ Game::Update()
//-------------------------------------------------------------------------------------
// Main update loop
//-------------------------------------------------------------------------------------
void
Game::UpdateObjects( float dt )
{
   // update player
   mPlayer->Update( dt );

}   // End of Game::Update()
// @ Game::Render()
//-------------------------------------------------------------------------------------
// Render stuff
//-------------------------------------------------------------------------------------
void
Game::Render()       // Here's Where We Do All The Drawing
{
   // set up viewer
   ::IvSetDefaultViewer( -10.f, 2.0f, 10.0f );

   // draw coordinate axes
   ::IvDrawAxes();

   // draw the main object
   mPlayer->Render();
}
```

Demo “Transformations 1” from Chapter 3

- Basic transformation code. Uses a full matrix and just concatenates new transforms.
- Controls
  - ESC: quit
  - I,J,K,L: translate in the XY plane
  - P: scale down, up
  - U,O: rotate around Z axis

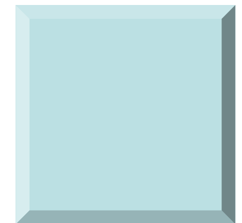
Routine for transformations 1

```
// @ Player::Update()
//-----------------------------------------------------------------------
// Main update loop
//-----------------------------------------------------------------------
void
Player::Update( float dt )
{
   // get change in transform for this frame
   IvMatrix44 scale, rotate, xlate;
   scale.Identity();
   rotate.Identity();
   float s = 1.0f;
   float r = 0.0f;
   float x = 0.0f, y = 0.0f, z = 0.0f;

   // set up scaling
   if (IvGame::mGame->mEventHandler-
     >IsKeyDown(';'))
   {
      s -= 0.25f*dt;
   }
   if (IvGame::mGame->mEventHandler-
     >IsKeyDown('p'))
   {
      s += 0.25f*dt;
   }
   scale.Scaling( IvVector3(s, s, s) );
```

```
// set up rotate
   if (IvGame::mGame->mEventHandler->IsKeyDown('o'))
   {
      r -= kPI*0.25f*dt;
   }
   if (IvGame::mGame->mEventHandler->IsKeyDown('u'))
   {
      r += kPI*0.25f*dt;
   }
   rotate.RotationZ( r );
    // set up translation
   if (IvGame::mGame->mEventHandler->IsKeyDown('k'))
   {
      x -= 3.0f*dt;
   }
   if (IvGame::mGame->mEventHandler->IsKeyDown('i'))
   {
      x += 3.0f*dt;
   }
   if (IvGame::mGame->mEventHandler->IsKeyDown('l'))
   {
      y -= 3.0f*dt;
   }
   if (IvGame::mGame->mEventHandler->IsKeyDown('j'))
   {
      y += 3.0f*dt;
   }
   IvVector3 xlatevector(x,y,z);
   xlate.Translation( xlatevector );

   // clear transform
   if (IvGame::mGame->mEventHandler->IsKeyDown(' '))
   {
      mTransform.Identity();
   }

   // append transforms for this frame to current transform
   // note order: mTransform is applied first, then scale, then rotate, then xlate
   mTransform = xlate*rotate*scale*mTransform;

}   // End of Player::Update()
```

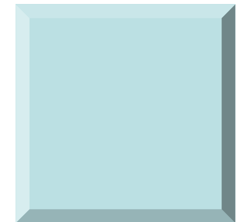
Demo “Transformations 2” from Chapter 3

- Basic transformation code. Undoes parts of transformation before concatenating changes so that resulting scales and rotations are centered on object.
- Controls
  - ESC: quit
  - I,J,K,L: translate in the XY plane
  - P: scale down, up
  - U,O: rotate around Z axis



Demo “Transformations 3” from Chapter 3

- Basic transformation code. Separates transformation into three parts and concatenates before rendering.
- Controls
  - ESC: quit
  - I,J,K,L: translate in the XY plane
  - P: scale down, up
  - U,O: rotate around Z axis

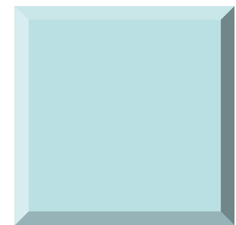


Demo "Transformations 4" from Chapter 3

- This code example shows a three-piece hierarchy showing how transforms concatenate.
- This shows a simple tank body-turret-barrel relationship.
- Controls
  - ESC: quit
  - I,J,K,L: translate tank in the XY plane
  - ;,P: scale tank down, up
  - U,O: rotate tank around Z axis
  - A,D: rotate turret left/right
  - W,S: raise barrel up/down
  - SPACE: reset transforms

Demo "Transformations 5" from Chapter 3

- This demo shows the power of a simple scene graph, which allows for quick and easy creation, management and rendering of hierachical scenes such as the articulated tank used herein. The demo allows the user to move and rotate the tank, which turns the turret and barrel as well. In addition, the turret and barrel can be independently articulated, as well
- The key commands are:
  - i, k - translate tank in x
  - j, l - translate tank in y
  - u, o - rotate tank around z
  - ;, p - scale tank
  - a, d - rotate turret around z
  - w, s - rotate barrel vertically
  - space - reset all transforms



Some of the most popular general purpose 3D modelers

- 3D Studio Max
- Alias
- Blender (open source)
- Cheetah3D
- Cinema 4D
- LightWave
- Maya
- MilkShape 3D
- modo
- Rhinoceros 3D
- Softimage|XSI
- trueSpace
- ZBrush

Some free modelers available via the Internet

- Anim8or
- Art of Illusion
- AutoQ3D
- K-3D
- Quake 2 Modeler
- ShapeShop
- SketchUp
- SmoothTeddy
- Wings 3D
- Zanoza Modeler