Overhead Slides for 3D Game Engine Design

Dave Eberly
Magic Software, Inc.

1 Introduction

The slides contained in this document were used in teaching a graduate Computer Science topics course at the University of North Carolina Chapel Hill in Spring 2001. The class met twice per week, each meeting lasting 1 hour and 15 minutes. The total number of lecture periods was approximately 30. Three of these lectures periods were devoted to visiting speakers from industry. No lectures were presented for the week of Spring break. The students either formed groups or worked as individuals on projects of their choice, subject to my approval. Each group or individual met with me twice during the semester in order to ensure progress on the project. Additionally, the students met with me individually for at most one hour for an oral examination, this designed to ensure attendance at the lectures (questions were on the concepts from the lectures). One day was reserved during final examination week so that the students could present their projects. Grading was based on the final projects and on oral examinations, each weighted 50 percent. The final project evaluation included comments about the presentation and included my feedback on the source code and its organization.

2 Course Layout

Relevant portions of the course layout are mentioned below.

Prerequisites: A general knowledge of computer graphics such as you would obtain in a senior level undergraduate course or an introductory graduate course. A background in vector and matrix algebra and some basic calculus of a single variable. Experience with an object–oriented programming language and with basic data structures.

Syllabus:

- Components of a game (not in book). Basic systems: storyline, game play, game AI, networking, graphics engine, content generation, etc.
- Chapter 3, sections 1,2,3 (transformations, camera model, screen space distance)
- Chapter 4 (hierarchical scene graphs)
- Appendix A (object–oriented infrastructure)
- Chapter 5 (picking = line–object intersection)
- Chapter 6 (collision = object–object intersection)
- Chapter 7, 8 (curves, surfaces; focus on subdivision)
- Chapter 9 (animation = time–varying quantities + interpolation)
- Chapter 13 (only particle systems and morphing, relative to animation). [Never covered particle systems.]
- Chapter 10, 11 (level of detail, terrain)
- Chapter 12 (spatial sorting) [Never covered this chapter.]
- Other topics as time permits (game physics, visibility and occlusion). [I managed a couple of lectures on physics.]

I know the book has large portions that are heavily mathematical. And the book is written in a dry style. Can mathematics be written any other way? I do not plan on stressing the mathematical issues, although this does not mean there will be no mathematical discussion in the course. There will be a lot of discussion about algorithms that are used in 3D graphics engines, including trade offs that must be considered based on the underlying hardware on which the engine will run.

Grading and Evaluation: It will be very difficult to build from scratch a 3D graphics engine, the game content, and a playable demo, all in one semester. In fact, some companies who have attempted to build a demo over a period of 3 to 6 months using NetImmerse for the purposes of convincing a publisher to fund their games have failed considerably.

Given that many of you had already planning your projects, I am willing to let you try so that you get a better appreciation of the complexity of building an entire game. I will venture to say that some folks have chosen not to sign up for this course because of the antipated time sink. In order to encourage those folks to reconsider, I propose the following alternatives for work required in the course:

- For those folks already certain they want to build the whole system, please continue. As warned, prepare to spend a lot of time on this project. I recommend using the source code that is included with the book. However, the book code does not include an exporter for any modeling package. You will need to figure out a way to export models–you can always use the ASCII export capabilities of modelers and write a simple importer.
- For those folks less inclined to participate in an ambitious project, I also propose that you use the source code that is included with the book. Your assignments will be to add a component to it (or multiple components, if the size and complexity is small for them) or enhance already existing components. The final project, so to speak, should be a basic demonstration that shows how well your components work and integrate into the engine. Some examples:
  - Build a basic collision system using hierarchical bounding volume trees.
  - Build a cloth animation system.
  - Enhance the OpenGL–based renderer to support new features. Some examples: The multitexturing hooks are in the code, but the full system needs to be developed. Add bump mapping and environment mapping support. Add support for projected lights and shadows.
  - Build a 3D spatialized sound system.
  - Build an exporter for 3D Studio Max, possibly including optimization tools to produce a scene graph that is structured to allow the engine to process it as rapidly and efficiently as possible.
  - Build an occlusion culling system.
  - Add a page–based terrain manager on top of the terrain system to allow predictive loading/unloading of terrain pages at run time.

  The list is somewhat endless. . .

In order to ensure progress and to give you feedback, each team (or individual if you choose not to work on a team) will meet with me three times during the semester. I will also suggest some problems over the course of the semester that you should take a look at. Part of the meetings will include discussions about how you would go about solving those problems. Each team (or individual if you choose not to work on a team) will present its final product to the class at the end of the semester.

Evaluation of your performance will be based on the meetings with me (50%) and the final product and presentation (50%).

3 The Slides

The slides are partitioned into 22 sections that correspond to the chunks in which I assembled them. I tended to generate the slides the night before the lectures. Listed below is a table indicating the dates on which

the files were created. Of course the actual dates are unimportant, but the difference in file creation times should give you an idea of the pace at which I covered the slides. Spring break occurred in the first week of March. The time difference between sections 11 and 12 indicates this. Each section has a cover page with a summary of topics and references to relevant book sections.

Section	Topic	Date	Page
1	coordinate systems and camera models	Jan 21	4
2	scene graph management	Jan 21	7
3	scene graph management	Jan 28	11
4	scene graph management	Jan 30	15
5	object-oriented infrastructure	Feb 04	19
6	object-oriented infrastructure	Feb 07	24
7	object-oriented infrastructure	Feb 18	27
8	intersections and collisions	Feb 18	30
9	intersections and collisions	Feb 20	35
10	intersections and collisions	Feb 25	40
11	intersections and collisions	Feb 27	44
12	curves	Mar 18	49
13	curves	Mar 20	53
14	surfaces	Mar 20	57
15	animation	Apr 08	61
16	animation	Apr 08	65
17	animation	Apr 16	68
18	level of detail	Apr 23	72
19	level of detail	Apr 24	79
20	game physics	Apr 29	86
21	game physics	Apr 30	90
22	game physics	May 02	95

Section 01. Coordinate Systems and Camera Models.

- Cartesian coordinates and other coordinate systems (2.2)
- Perspective projection (3.2)
- Standard perspective and general camera models (3.3)
- Mapping from model space to screen space (3.3)

• **Standard Cartesian coordinates.** origin $\vec{0} = (0,0,0)$; axes $\vec{\imath} = (1,0,0)$, $\vec{\jmath} = (0,1,0)$, $\vec{k} = (0,0,1)$; points are $(x,y,z) = x\vec{\imath} + y\vec{\jmath} + z\vec{k}$. The components are ordered. This system is right–handed, $\vec{k} = \vec{\imath} \times \vec{\jmath}$.

• **Other coordinate systems.** origin $\vec{P}$; axes are $\vec{U}_0$, $\vec{U}_1$, $\vec{U}_2$. The axes are mutually orthonormal. Set $R = [\vec{U}_0\,\vec{U}_1\,\vec{U}_2]$, columns of R are the axes. R is orthogonal. If $\det(R) = 1$, coordinates are right–handed. If $\det(R) = -1$, coordinates are left–handed. Points are $\vec{X} = \vec{P} + R\vec{Y}$. $\vec{X}$ is Cartesian triple, $\vec{Y}$ is the representation of $\vec{X}$ in the coordinate system, determined by $\vec{Y} = R^{\mathrm{T}}(\vec{X} - \vec{P})$, or $y_i = \vec{U}_i \cdot (\vec{X} - \vec{P})$.

• **Perspective projection.** Decide what to draw in the world, convert to screen coordinates, send to renderer. Eye point is $\vec{E}$ and view plane is $\vec{N} \cdot \vec{X} = d$ with $\vec{N} \cdot \vec{E} > d$ (eye point on "positive side" of view plane). Perspective projection of $\vec{X}$ onto view plane is intersection of plane and ray starting at $\vec{E}$ and containing $\vec{X}$. Projection is $\vec{Y} = \vec{E} + t(\vec{X} - \vec{E})$ where

$$t = \frac{\vec{N} \cdot \vec{E} - d}{\vec{N} \cdot (\vec{E} - \vec{X})}.$$

• **Standard perspective camera model.** Camera is at eye point $(0,0,0)$. View plane is $z = n > 0$, called the near plane. Points far away are not considered, so limit $z \le f$. Far plane is $z = f$. Only projections onto a viewport are allowed, so limit $l \le x \le r$ and $b \le y \le t$. The region bounded by these constraints is the view frustum. Typically frustum is "orthogonal", $l = -r$ and $b = -t$.

Camera has direction of view $\vec{D} = (0,0,1)$ and "up" vector $\vec{U} = (0,1,0)$. Need one more vector $\vec{L}$ so that camera has coordinate system with origin $\vec{E}$ and axes $\vec{L}$, $\vec{U}$, and $\vec{D}$. Choose $\vec{L} = (1,0,0)$ so that axes, in order specified, forms a right–handed system. Point (x,y,z) inside view frustum projects to near plane point $(nx/z, ny/z, n) = (x/w, y/w, n)$ where $w = z/n$.

Axis of frustum contains eye point and center of frustum, $((r+l)z/(2n), (t+b)/(2n), z)$ for $z \in [n,f]$. Map frustum to orthogonal frustum with viewport $[-1,1]^2$. Do this by

$$x' = \frac{2}{r-l}\left(x - \frac{(r+l)z}{2n}\right), \quad y' = \frac{2}{t-b}\left(y - \frac{(t+b)z}{2n}\right)$$

Transform z values to $[0,1]$ by

$$z' = \frac{f}{f-n}\left(1 - \frac{n}{z}\right).$$

Screen has pixels (x'', y'') with $0 \le x'' < S_x$ and $0 \le y'' < S_y$ and are left–handed (y'' increases from top row to bottom row, x'' increases from left column to right column). Final conversion to screen coordinates is

$$x'' = \frac{(S_x - 1)(x' + 1)}{2}, \quad y'' = \frac{(S_y - 1)(1 - y')}{2}.$$

• **General camera model.** Right–handed coordinate system for camera has origin $\vec{E}$; axes $\vec{L}$, $\vec{U}$, $\vec{D}$. View plane has origin $\vec{P} = \vec{E} + n\vec{D}$. Viewport is defined by the four corners $\vec{P} + \{r, l\}\vec{L} + \{t, b\}\vec{U}$. World point $\vec{X}$ is related to camera coordinates $\vec{Y}$ by $\vec{X} = \vec{E} + R\vec{Y}$ where $R = [\vec{L}\,\vec{U}\,\vec{D}]$, so $\vec{Y} = R^{\mathrm{T}}(\vec{X} - \vec{E})$. This is the view tranformation.

• **Mapping from model coordinates to the screen.** Transform object from model coordinates to world coordinates. Transform from world coordinates to camera coordinates. Project points onto view plane. Convert to screen coordinates. (Discuss as homogeneous matrices.)

Section 02. Scene Graph Management.

- Scene graph management (4)
- Hierarchical transforms, local and world (4.1.1)
- Hierarchical culling (4.1.2, 4.3)
- Geometric update of scene (4.2)

- **Scene graph management.** For organization of the audible and visual data in the world. Typically the organization is hierarchical.
  - Allows construction of complex objects from simple objects. Articulated objects, for example, humanoid characters. Placement of objects within a level. Notions of model, local, and world transforms.
  - Allows rapid culling of nonvisible objects–minimize the data sent to the renderer. Notions of bounding volumes (model and world) and intersection testing between bounding volumes and view frustum.
  - Allows specification of render state at a single node in the hierarchy that affects an entire subtree. Indirectly supports drawing based on sorted render states.
  - Notion of controller for time–varying transforms and other data.
  - Shared data makes the management more complicated. Sharing via a directed acyclic graph (implicit sharing). Sharing of the raw data (explicit sharing).

- **Hierarchical Transforms.** Homogeneous transform without perspective projection is
$$\vec{Y} = \left\langle M \mid \vec{T} \right\rangle \vec{X} = M\vec{X} + \vec{T}.$$
Product is
$$\left\langle M_1 \mid \vec{T}_1 \right\rangle \left\langle M_2 \mid \vec{T}_2 \right\rangle = \left\langle M_1 M_2 \mid M_1\vec{T}_2 + \vec{T}_1 \right\rangle.$$
Assuming M is invertible, inverse is
$$\left\langle M \mid \vec{T} \right\rangle^{-1} = \left\langle M^{-1} \mid -M^{-1}\vec{T} \right\rangle.$$
Allowing general M (rotation, scaling, shearing) confounds the situation. How to determine the primitive components? Inversion of 3×3 can be expensive. Support only rotation R, translation $\vec{T}$, and uniform scaling s. Transform is $\langle sR|\vec{T}\rangle$. Inverse is
$$\left\langle sR \mid \vec{T} \right\rangle^{-1} = \left\langle \frac{1}{s}R^{\mathrm{T}} \mid -\frac{1}{s}R^{\mathrm{T}}\vec{T} \right\rangle.$$
Equation $\vec{Y} = sR\vec{X} + \vec{T}$ inverts to $\vec{X} = R(\vec{Y} - \vec{T})/s$.

- **Local and World Transforms.** Local transform at a node specifies how the node is positioned with respect to its parent. Given parent node P with child node C, the world transform of C is

$$\begin{aligned}\left\langle M^{(C)}_{\text{world}} \mid \vec{T}^{(C)}_{\text{world}} \right\rangle &= \left\langle M^{(P)}_{\text{world}} \mid \vec{T}^{(P)}_{\text{world}} \right\rangle \left\langle M^{(C)}_{\text{local}} \mid \vec{T}^{(C)}_{\text{local}} \right\rangle \\ &= \left\langle M^{(P)}_{\text{world}} M^{(C)}_{\text{local}} \mid M^{(P)}_{\text{world}} \vec{T}^{(C)}_{\text{local}} + \vec{T}^{(P)}_{\text{world}} \right\rangle .\end{aligned}$$

  (Draw picture illustrating this.)

- **Hierarchical Culling.** Determination if data is in the view frustum by testing a triangle at a time is expensive. If data is represented hierarchically, then attempt to cull large portions at a time by using bounding volumes. The idea is that testing a bounding volume against the frustum is inexpensive. The amortized cost of bounding volume comparisons and the reduced amount of drawing objects outside the frustum is less than drawing all triangles. Issues:
  - Which bounding volume to choose? Trade off between accurate fit of data by volume and cost of comparing volume to frustum.
  - Use plane–at–a–time culling or completely accurate comparison? First method is fast, but can lead to objects being drawn that are not in frustum. Second method is slower, but only objects inside (or partially inside) frustum are drawn. Trade off depends on the data.
  - Leaf nodes have model bounding volume. (Static construction off–line.) Transforms place the data in the world, so also need world bounding volume. Each node in scene graph needs world bounding volume. How to build them? (Dynamic construction at run time.)

- **Geometric Update of Scene Tree.** The process is the following. The idea is that transforms are propagated down the tree and bounding spheres are propagated up the tree.
  - Local transforms at various nodes are changed.
  - World transforms at all affected nodes must be recalculated. Affected nodes are in subtrees of nodes at which local transforms changed. Recalculate by recursive traversal. Can recalculate starting at root,

but better is to maintain minimum set of nodes at which traversal is started.

- At leaf nodes, world transforms are now known. World bounding volumes must be calculated by transforming model bounding volumes. If model data was changed (morphing, skinning, etc.), then model bounding volume needs to be updated first.
- On recursive return, parent node now knows all world bounding volumes for child nodes. Compute world bounding volume that contains all the child world bounding volumes.
- When recursion returns to calling node, that node's world bounding volume is usually different than before the call. You need to propagate this change to the root of the tree.

Section 03. Scene Graph Management.

- Scene tree data structures (classes in book source code)
- Geometric state update functions (4.2)
- Render state update functions (4.2)

- **Scene tree structure.** `Spatial` represents transforms and bounds, has parent links. Two types of derived classes. `Node` adds child links and recursive traversal. `Geometry` adds visual data and repository for visual render state. If 3D sound is part of system, `Sound` adds audio data and repository for audio render state. Update of geometric state:

```
void Spatial::UpdateGS (float time, bool initiator)
{   // virtual: Spatial handles transforms
    //          Geometry-derived might do some work
    //          Node handles recursion
    UpdateWorldData(time);

    // virtual: Geometry does model-to-world
    //          Node does merging of bounds
    UpdateWorldBound();
    if ( bInitiator ) PropagateBoundToRoot();
}

void Node::UpdateWorldData (float time)
{
    Spatial::UpdateWorldData(time);
    for each child do
        child.UpdateGS(time,false);
}

void Node::UpdateWorldBound ()
{
    worldBound = firstChild.worldBound;
    for each additional child do
        Merge(worldBound,child.worldBound);
}

void Geometry::UpdateWorldBound ()
{  // derived class might require modelBound recalc
   worldBound = modelBound.TransformBy(worldTransform);
}
```

```
void Spatial::UpdateWorldData (float time)
{
    // Update spatial controllers.  Examples are
    // keyframe, inverse kinematics, skinning, morphing,
    // particle systems.
    bool computesWorldTransform = false;
    for each controller do
        computesWorldTransform = controller.Update(time);

    // update world transforms
    if ( !computesWorldTransform )
    {
        if has parent then
        {
            worldTransform =
                parent.worldTransform*localTransform;
        }
        else
        {
            worldTransform = localTransform;
        }
    }
}
```

- **Render state updating.** Each leaf node in scene tree maintains the render state that affects it. If render state attached to interior node, it must be propagated to leaves. If done as shallow copy, then render state update only required on attach/detach of state or topology changes to tree.

```
void Spatial::UpdateRS (bool initiator)
{
    // update previous state by current state
    if ( initiator )
    {
        PropagateStateFromRoot();
    }
    else
    {
        for each renderState do
            RS.Push(renderState);
    }

    // virtual:  Geometry makes shallow copy
    //           Node handles recursion
    UpdateRenderState();

    // restore previous state
    if ( initiator )
    {
        RestoreStateToRoot();
    }
    else
    {
        for each renderState do
            RS.Pop(renderState);
    }
}
```

Section 04. Scene Graph Management.

- Render state update functions (4.2.7)
- Drawing the scene tree (4.3.7)

```
void Node::UpdateRenderState ()
{
    for each child do
        child.UpdateRS(false);  // not the initiator
}

void Geometry::UpdateRenderState ()
{
    // shallow copy current render state to local state
    RS.CopyTo(state);
}
```

- **Rendering the scene.** `Renderer` represents the rendering system that contains a `Camera` (coordinate system, view frustum parameters). The entry point to drawing a scene is the renderer's `Draw` method and starts a recursive traversal of the scene.

```
void Renderer::Draw (Node scene)
{
    scene.OnDraw(self);
}
```

```
void Spatial::OnDraw (Renderer renderer)
{
    if ( forceCulling ) return;

    Camera camera = renderer.GetCamera();

    // Camera does plane-at-a-time culling.  Frustum
    // planes can be turned off for child culling if
    // the parent is "inside" the plane.
    PlaneState state = camera.GetPlaneState();

    if ( not camera.Culled(worldBound) )
        Draw(renderer);

    // If planes were turned off, they must be turned
    // back on.
    camera.SetPlaneState(state);
}

void Node::Draw (Renderer renderer)
{
    for each child do
        child.OnDraw(renderer);
}

void Geometry::Draw (Renderer renderer)
{
    renderer.SetState(state);
}

void GeometryDerived::Draw (Renderer renderer)
{
    Geometry::Draw(renderer);
    renderer.Draw(self);  // Draw needed per derived class
}
```

- **Deferred drawing to allow sorting.** The `Renderer` object can allow a render-state sorting function to be specified. If such a function is specified, the renderer places visible objects on a list to be drawn later.

```
void Renderer::SetState (RenderState state)
{
    if ( not hasSorter )
        renderer.SetState(state);
}

void Renderer::Draw (GeometryDerived object)
{
    if ( hasSorter )
    {
        AddToDeferredDrawingList(object);
        return;
    }
    <code for drawing GeometryDerived objects>;
}

void Renderer::Draw (Node scene)
{
    scene.OnDraw(self);
    if ( hasSorter )
    {
        SortDeferredDrawingList();
        hasSorter = false;
        for each object in list do object.Draw(self);
        hasSorter = true;
        RemoveAllFromDeferredDrawingList();
    }
}
```

Section 05. Object–Oriented Infrastructure.

- Object–Oriented Issues (Appendix A)
- Inheritance and Run Time Type Informatoin (A.3)

- **Object–Oriented Issues.**
  - **Run–time type information (RTTI).** When an object is known only through a base class pointer, determine the class of the object. Determine that the class of an object is derived from another class. Queries for other class information may also be supported.
  - **Shared objects and reference counting.** Dynamically allocated objects can be referenced by two or more objects. Keep track of the number of references to an object. When an object loses its last reference, automatically delete it. The system that handles the reference counting uses objects called *smart pointers.*
  - **Templates in a shared object system.** Standard template libraries can be used for basic data structures (arrays, lists, stacks, queues, maps, etc.). Care must be taken to ensure side effects are properly handled.
  - **Streaming.** Application data is an abstract graph of objects. Need to read (write) data from (to) disk, memory, or across network.
  - **Startup and shutdown.** Various systems must initialize before the main program begins its work (*pre–main startup*) and terminate after the main program ends its work (*post–main shutdown*).
  - **Namespaces, naming conventions, programming standards.** Yes, these are important in a product, whether or not you provide source code to customers.

  An object–oriented library supporting such systems requires a base class (or base classes) to host the systems. The propagation of the systems through derived classes is usually best supported by macros. In the book source, the base class is `Object`.

- **Inheritance Systems.** The class system is a directed graph where each nodes represents a class and each directed arc from a node D to a node B indicates that D is a class derived from a base class B. A *single–inheritance* class system is one where each connected component of the graph is a tree. Otherwise, the system is a *multiple–inheritance* class system. Regardless, each class has a unique object that stores information about that class (a RTTI object).

- **RTTI and single inheritance.** Easy to support. For a single class tree, each RTTI object stores (1) information for the class and (2) a link to the RTTI object of its immediate base class. (In book source, class information is only a string containing the name of the class.)

  Given an object pointer, it can be *upcast* to a pointer of any class type along the path from object class to root class. This is always safe (in single–inheritance systems). Given a base class pointer to an object, it can be *downcast* to a pointer of any class type along the path from base class to object class. Multiple–inheritance systems also support *crosscast* (casting across base classes of a derived class). The process generally is called *dynamic casting*. The result of the cast is `NULL` if the casting is invalid. Otherwise, the *correct* pointer is returned. In multiple–inheritance systems, the result of the cast is *not necessarily the original pointer*.

  Dynamic casting systems built into a compiler tend to be slow because they must support multiple inheritance and a class system that is not a single connected graph. For a specialized system (such as your own real–time graphics system), it is better to implement your own RTTI and use a single–inheritance system.

```
#define DeclareRootRTTI
public:
    static const RTTI ms_kRTTI;

    virtual const RTTI* GetRTTI () const
    {  return &ms_kRTTI; }

    bool IsExactlyClass (const RTTI* pkQueryRTTI) const
    {  return ( GetRTTI() == pkQueryRTTI ); }

    bool IsDerivedFromClass (const RTTI* pkQueryRTTI) const
    {
        const RTTI* pkRTTI = GetRTTI();
        while ( pkRTTI )
        {
            if ( pkRTTI == pkQueryRTTI ) return true;
            pkRTTI = pkRTTI->GetBaseRTTI();
        }
        return false;
    }

    void* DynamicCast (const RTTI* pkQueryRTTI)
    {  return ( IsDerivedFromClass(pkQueryRTTI) ? this : 0 ); }

#define DeclareRTTI
public:
    static const RTTI ms_kRTTI;

    virtual const RTTI* GetRTTI () const
    { return &ms_kRTTI; }
```

These macros hide the RTTI class.

```
#define ImplementRootRTTI(rootclassname)
    const RTTI rootclassname::ms_kRTTI(#rootclassname,NULL)

#define ImplementRTTI(classname,baseclassname)
    const RTTI classname::ms_kRTTI(#classname,
        &baseclassname::ms_kRTTI)

#define IsExactlyClass(classname,pObject)
    ( pObject
      ?
      pObject->IsExactlyClass(&classname::ms_kRTTI)
      :
      false )

#define IsDerivedFromClass(classname,pObject)
    ( pObject
      ?
      pObject->IsDerivedFromClass(&classname::ms_kRTTI)
      :
      false )

#define StaticCast(classname,pObject)
    ((classname*)(void*)pObject)

#define DynamicCast(classname,pObject)
    ( pObject
      ?
      (classname*)pObject->DynamicCast(&classname::ms_kRTTI)
      :
      NULL )
```

Section 06. Object–Oriented Infrastructure.

- Sharing objects and reference counting (A.5)
- Smart pointers (A.5)

- **Shared objects and reference counting.** Some issues are
  - Implicit system (smart pointers) or explicit system (global memory manager)? Either system keeps a *reference count* per object and routines to increment/decrement the count. When count becomes zero, object should be destroyed.
  - Side effects with smart pointers (example, implicit constructor/destructor calls).
  - Cycles can prevent object destruction (cannot force reference count to zero).
  - Object leaks (tools that track memory leaks will miss these).

```
#define DeclareRootSmartPointers
public:
    void IncrementReferences ()
    {
        m_uiReferences++;
    }

    void DecrementReferences ()
    {
        if ( --m_uiReferences == 0 )
            delete this;
    }

    unsigned int GetReferences ()
    {
        return m_uiReferences;
    }

private:
    unsigned int m_uiReferences;

#define SmartPointer(classname)
class classname;
typedef Pointer<classname> classname##Ptr
```

```
template <class T>  // T must be Object or Object-derived
class Pointer
{
public:
    Pointer (T* pkObject = 0);  // increments
    Pointer (const Pointer& rkPointer);  // increments
    ~Pointer ();  // decrements

    // implicit conversions (get regular pointer behavior)
    operator T* () const;
    T& operator* () const;
    T* operator-> () const;

    // assignment (increment/decrement)
    Pointer& operator= (const Pointer& rkReference);
    Pointer& operator= (T* pkObject)
    {   if ( m_pkObject != pkObject )
        {
            if ( m_pkObject )
                m_pkObject->DecrementReferences();
            m_pkObject = pkObject;
            if ( m_pkObject )
                m_pkObject->IncrementReferences();
        }
        return *this;
    }

    // comparisons (get regular pointer behavior)
    bool operator== (T* pkObject) const;
    bool operator!= (T* pkObject) const;
    bool operator== (const Pointer& rkReference) const;
    bool operator!= (const Pointer& rkReference) const;

protected:
    T* m_pkObject;  // the shared object
};
```

Section 07. Object–Oriented Infrastructure.

- Streaming of scene graphs (A.6)

- **Streaming of Scene Graphs.** A scene graph can be viewed as a directed graph whose nodes represent objects of type `Object` and whose directed arcs represent `Object` pointers. A class `Stream` has the job of managing the streaming.

  To save a scene graph to disk or memory:

  - **Register.** Traverse the parent–child links of the graph to create a unique list of objects. Each `Spatial` object adds itself to list (if not already in list) and tells each `Object` member to add itself to list. Beware of cycles (okay to have). Beware of multiple registration of object (can prevent).
  - **Save.** Traverse the list of unique objects and tell each one to save itself.
    * Write RTTI identifier. (`Object`)
    * Write object pointer as a unique ID. (`Object`)
    * Tell base class to save itself.
    * Write your native data (such data does not know how to save itself). No need to Write "derived" data.
    * Write your object pointers to serve as unique IDs for loading and linking.

  Loading a scene graph from disk or memory is more complicated. `Stream` must load "old" pointers (unique IDs), associate them with "new" pointers (newly created objects), and resolve addresses *after* everything is loaded. That is, the streaming system is both a *loader* and a *linker*. `Stream` manages a list of pairs `(UniqueID,Link)` for the linking phase where `Link` minimally stores the new pointer.

  - **Factory.** The type of the to–be–loaded object must be known *before* you load it so you can create a new object (in memory) and load/set its data members. `Stream` reads the RTTI identifier, looks up the corresponding factory function that knows how to load an object of the given type, then calls the factory function to create a new object.
  - **Load.**
    * Tell base class to load itself. Non–abstract classes create a `Link` object and pass it to base class calls of `Load`. At root class `Object`,

Unique ID (was old object pointer) read by `Object`, added to the `Link` object, and `(UniqueID,Link)` pair given to `Stream`.

    * Read your native data (such data does now know how to load itself).
    * Create "derived" data from native data.
    * Read your unique IDs (were old object pointers) and store them in `Link` object.

- **Link.** After loading, `Stream` has a list of memory objects whose `Object` pointer members (stored separately in the `Link` objects) must be replaced by the corresponding memory address of the newly created objects. `Stream` iterates over list of objects, gives each object its `Link` object, then tells object to link itself:
    * Tell base class to link itself.
    * Resolve your object pointer members.
- **Post–link Semantics.** Be careful to make sure that the link calls do not require obtaining information from other objects. For example, the `Node` link call should not make an `UpdateGS` call since you have no idea of link order. Objects in the subtree rooted at the node might not have been linked at the time of the call. Reserve all cross–object interactions until after the linking phase.

Saving and loading multiple scene graphs requires `Stream` to maintain a list of "top–level objects". If you save two scenes to file, you want to load file and have access to the two scenes. The streaming system must provide a sentinel to identify the top–level objects.

Cross–platform support requires handling issues such as byte order, floating point representation, and size of things such as Boolean type and enumerated types.

Section 08. Intersections and Collisions.

- Intersection queries (5,6)
- Picking (5)
- Culling (4.3)
- Collision (6)
- Bounding volume trees (6.6, but more general)

- **Intersection Queries.** A simplistic classification scheme is
  - **Picking:** Intersection of line and (solid) object.
  - **Culling:** Intersection of plane and (solid) object.
  - **Collision:** Intersection of two (solid) objects.

  Many folks throw all of these under the general topic of *collision detection*.

- **Two types of queries.**
  - **Test.** Determine *if* two geometric objects intersect.
  - **Find.** Determine *where* two geometric objects intersect.

  The *test* query is usually less expensive than the *find* query. For example, consider a line and a sphere. The line intersects the sphere if the distance from sphere center to line is smaller or equal to sphere radius (can be done with a few arithmetic operations). If the line intersects the sphere, the points of intersection can be constructed by solving a quadratic equation (involves an expensive square root operation).

- **Intersection of moving objects.** The *test* query is usually more complicated for moving objects than for stationary objects. The *find* query now involves computing the *first time of contact* and the *intersection set at first contact* (sometimes called the *contact manifold*).

  A powerful method for handling intersection of convex polyhedra is the *method of separating axes* (projection onto lines; separation of intervals).

- **The computer graphics N–body problem.** Given N objects, find all intersections between them. There are $N(N-1)$ pairs, so for large N the cost of intersection testing can be enormous. How to reduce the cost?
  - **Collision groups (large scale).** Partition the objects into small groups and detect intersections only between objects within a group.
  - **Hierarchical decomposition (small scale).** Partition each object into some hierarchical structure. The idea is to *localize* the search by quickly rejecting portions of the objects that do not intersect.

- **Collision response (game physics).** What to do when two objects intersect? (Bounce off wall, slide along wall, blow up, bend, deform, etc.)

- **Picking.** The classic problem for which the method is named is to select (via a mouse) an object shown on the 2D screen and have the program find that object in the world data base.

  Read screen coordinates at mouse click. Compute corresponding point in viewport on near plane (a point in the world). Create ray from eye point passing through the viewport point. Determine which objects in the world are intersected by that ray.

  An example not related to mouse clicks. Character fires laser gun at objects. Determine if objects actually hit by laser.

  In a hierarchical scene graph, you can use bounding volumes to help localize the search.

```
void DoPick (Node node, Ray ray, PickResults results)
{
    if ( ray intersects node.boundingVolume ) then
    {
        if ( node is interior ) then
        {
            for ( each child of node ) do
                DoPick(child,ray,results);
        }
        else  // node is leaf geometry
        {
            for ( each triangle of node ) do
                add intersection of ray-triangle to results;
        }
    }
}
```

  The actual results can be processed during the pick operation or postprocessed to handle issues of selection of specific attributes (intersection location; color, texture coordinates, normal at point of intersection; sorting along ray).

- **Culling.** The classic problem is to determine if object (or bounding volume) is partially or fully inside the view frustum. Plane–at–a–time culling involves testing if the object/BV is outside any plane containing a face of the frustum.

  An example not related to culling. Character walks around in room. Make sure it does not walk through a wall. Another example–sort a scene by binary separating planes (BSP trees).

  The *test* query can be performed easily by projection onto a normal line to the plane. The *find* query has complexity related to that of the object of the query. Examples in 2D: convex polygon, ellipse (will discuss in class in some detail).

- **Collision.** The most difficult of the problems. The prototypical hierarchical approach for comparing two triangle meshes is OBB trees (more generally, *bounding volume trees*). This is a top–down approach. Construction of binary tree involves:

  - Construct bounding volume for mesh.
  - Project triangles (or some related data such as center of mass) onto a special line.
  - Use some statistic on the projection to partition triangles into two submeshes (split by average, split by median).
  - Repeat the process on the two submeshes.

Comparison of two BV trees for intersection involves double recursion:

```
void IntersectionQuery (Tree T0, Tree T1)
{   if ( BV(T0) intersects BV(T1) )
    {   if ( T0 is interior node )
        {   IntersectionQuery(T0.LChild,T1);
            IntersectionQuery(T0.RChild,T1);
        }
        else if ( T1 is interior node )  // T0 is leaf
        {   IntersectionQuery(T0,T1.LChild);
            IntersectionQuery(T0,T1.RChild);
        }
        else  // T0 and T1 are leaves
        {   for ( each triangle G0 of T0 ) do
            {   for ( each triangle G1 of T1 ) do
                {   if ( G0 and G1 intersect ) then
                        report intersection results;
                }
            }
        }
    }
}
```

Complications.

- BV Tree is constructed in *model space*. The objects are in world space, so bounding volumes must be transformed from model space to world space. (Discuss space–versus–time trade off.)
- How to report results?
- What if model space geometry changes? (Object not moving rigidly.)

Section 09. Intersections and Collisions.

- Method of separating axes (Various parts in book, but really new material to appear in the Geometric Methods book. PDF document to use is `MethodOfSeparatingAxes.pdf`

- **Method of Separating Axes.** Two convex objects do not intersect if and only if there exists a line for which the projections of the objects onto the line do not intersect. Such a line is called a *separating axis*.
  - Sufficient to consider lines through the origin.
  - If projection intervals of the convex objects C_i are $[\lambda^{(i)}_{\min}, \lambda^{(i)}_{\max}]$, then the separation test is:

$$\lambda^{(0)}_{\max} < \lambda^{(1)}_{\min} \ \text{ or } \ \lambda^{(1)}_{\max} < \lambda^{(0)}_{\min}.$$

  - Sufficient to consider $\vec{D}$ and not $-\vec{D}$. Not important that $\vec{D}$ be unit length.
  - The set of potential separating axes for convex polygons (2D,3D) and convex polyhedra (3D) is finite. (In 2D, illustrate with axis–aligned rectangles, oriented rectangles, triangles. In 3D illustrate with axis–aligned boxes, oriented boxes, triangles.)
  - Separation of non–polygonal and non–polyhedral objects is typically more complicated. (In 2D, illustrate with ellipses. In 3D, illustrate with ellipsoids and cylinders.)

Direct implementation is

```
void ComputeInterval (ConvexPolygon C, Point D,
    float& min, float& max)
{
    max = min = Dot(D,C.V(0));
    for (i = 1; i < C.N; i++)
    {   value = Dot(D,C.V(i));
        if ( value < min ) min = value;
        else if ( value > max ) max = value;
    }
}
```

```
bool TestIntersection (ConvexPolygon C0, ConvexPolygon C1)
{
    // test edge normals of C0 for separation
    for (i1 = 0; i0 = C0.N-1; i1 < C0.N; i0 = i1, i1++)
    {   D = Perp(C0.V(i1) - C0.V(i0));
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        if ( max1 < min0 || max0 < min1 )
            return false;
    }

    // test edge normals of C1 for separation
    for (i1 = 0; i0 = C1.N-1; i1 < C1.N; i0 = i1, i1++)
    {   D = Perp(C1.V(i1) - C1.V(i0));
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        if ( max1 < min0 || max0 < min1 )
            return false;
    }

    return true;
}
```

More efficient implementation is

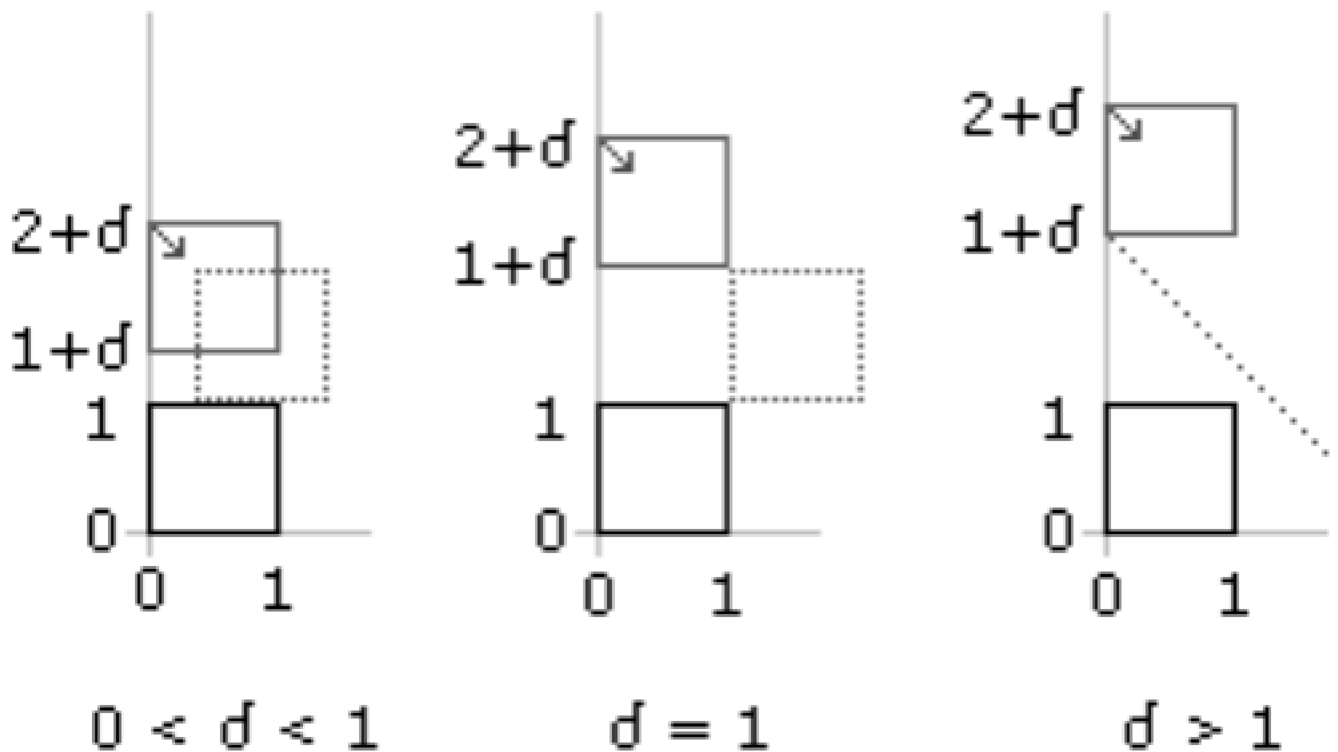
```
int GetMiddleIndex (int i0, int i1, int N)
{
    if ( i0 < i1 )
        return (i0 + i1)/2;
    else
        return (i0 + i1 + N)/2 % N );
}

int GetExtremeIndex (ConvexPolygon C, Point D)
{
    i0 = 0; i1 = 0;
    while ( true )
    {
        iM = GetMiddleIndex(i0,i1);
        iNext = (iM + 1) % N;
        E = C.V(iNext) - C.V(iM);
        if ( Dot(D,E) > 0 )
        {
            if ( iM != i0 ) i0 = iM; else return i1;
        }
        else
        {
            iPrev = (iM + N - 1) % N;
            E = C.V(iM) - C.V(iPrev);
            if ( Dot(D,E) < 0 ) i1 = iM; else return iM;
        }
    }
}
```

```
bool TestIntersection (ConvexPolygon C0, ConvexPolygon C1)
{
    // Test C0 edges for separation.  Because of ccw
    // ordering, projection interval for C0 is [m,0], m <= 0.
    // Try to determine if C1 on positive side of line.
    for (i1 = 0, i0 = C0.N-1; i1 < C0.N; i0 = i1, i1++)
    {
        D = Perp(C0.V(i1) - C0.V(i0));
        iMin = GetExtremeIndex(C1,-D);
        diff = C1.V(iMin) - C0.V(i1);
        if ( Dot(D,diff) > 0 )
        {   // C1 entirely on positive side of C0.V(i0)+t*D
            return false;
        }
    }

    // Test C1 edges for separation.  Because of ccw
    // ordering, projection interval for C1 is [m,0], m <= 0.
    // Try to determine if C0 on positive side of line.
    for (i1 = 0, i0 = C1.N-1; i1 < C1.N; i0 = i1, i1++)
    {
        D = Perp(C1.V(i1) - C1.V(i0));
        iMin = GetExtremeIndex(C0,-D);
        diff = C0.V(iMin) - C1.V(i1);
        if ( Dot(D,diff) > 0 )
        {   // C0 entirely on positive side of C1.V(i0)+t*D
            return false;
        }
    }

    return true;
}
```

Section 10. Intersections and Collisions.

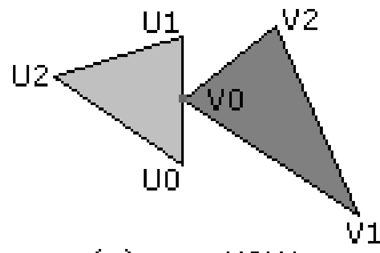
- Separation of moving convex polyhedra (`MethodOfSeparatingAxes.pdf`)
- First time of contact
- Intersection set at first time of contact

- **Separation of Moving Convex Polyhedra.** Given two moving convex polyhedra that are not initially intersecting:
  - If the first intersection is at time $T_{\text{first}} > 0$, then their projections along every line must intersect at that time. An instant before first time of contact, there must be at least one separating axis.
  - If the last intersection is at time $T_{\text{last}} > 0$, then their projections along every line must intersect at that time. An instant after last time of contact, there must be at least one separating axis.
  - Compute first and last contact times of the polyhedra by computing first and last contact times of the projection intervals. T_{first} is computed as the largest time for which there is at least one separating axis for each $t \in (-\infty, T_{\text{first}}]$. T_{last} is computed as the smallest time for which there is at least one separating axis for each $t \in [T_{\text{last}}, \infty)$.
  - If $T_{\text{first}} \le T_{\text{last}}$, the polyhedra intersect if and only if $t \in [T_{\text{first}}, T_{\text{last}}]$. It is possible in the construction that $T_{\text{first}} > T_{\text{last}}$, in which case the objects never intersect.

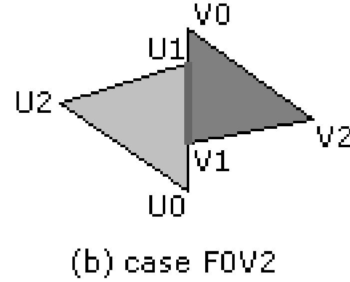


```
bool TestIntersection (Convex C0, Convex C1, float TMax,
    float& TFirst, float& TLast)
{   W = C1.W - C0.W;  // C0 is stationary, C1 is moving
    S = Union(C0.S,C1.S);  // all potential separating axes
    TFirst = -INFINITY;  TLast = INFINITY;
    for each D in S do
    {   speed = Dot(D,W);
        ComputeInterval(C0,D,min0,max0);
        ComputeInterval(C1,D,min1,max1);
        if ( max1 < min0 )  // C1 initially on 'left' of C0
        {   if ( speed <= 0 ) return false;  // moving apart
            T = (min0 - max1)/speed;  if ( T > TFirst ) TFirst = T;
            if ( TFirst > TMax ) return false;  // 'quick out'
            T = (max0 - min1)/speed;  if ( T < TLast ) TLast = T;
            if ( TFirst > TLast ) return false;  // 'quick out'
        }
        else if ( max0 < min1 )  // C1 initially on 'right' of C0
        {   if ( speed >= 0 ) return false;  // moving apart
            T = (max0 - min1)/speed;  if ( T > TFirst ) TFirst = T;
            if ( TFirst > TMax ) return false;  // 'quick out'
            T = (min0 - max1)/speed;  if ( T < TLast ) TLast = T;
            if ( TFirst > TLast ) return false;  // 'quick out'
        }
        else  // interval(C0) and interval(C1) overlap
        {   if ( speed > 0 )
            {   T = (max0 - min1)/speed;  if ( T < TLast ) TLast = T;
                if ( TFirst > TLast ) return false;  // 'quick out'
            }
            else if ( speed < 0 )
            {   T = (min0 - max1)/speed;  if ( T < TLast ) TLast = T;
                if ( TFirst > TLast ) return false;  // 'quick out'
            }
        }
    }
    return true;
}
```

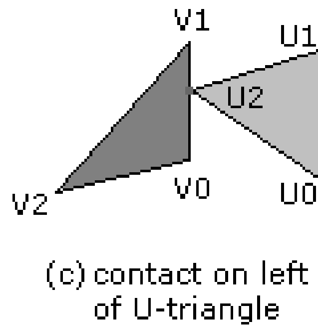
- **Intersection Set at First Time of Contact.** Generally tedious to code, but still tractable. In addition to computing first time of contact, keep track of which vertices project to the extremes. This information tells you the orientation of the two convex objects relative to each other at the time of contact: vertex–vertex, vertex–edge, edge–edge (2D or 3D), vertex–face, edge–face, face–face (3D).



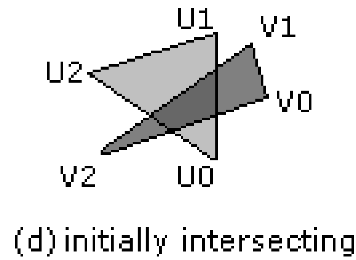
(a) case V0V1



(b) case F0V2



(c) contact on left of U-triangle



(d) initially intersecting

SECTION 11. INTERSECTIONS AND COLLISIONS.

- Finding the contact set (`MethodOfSeparatingAxes.pdf`)

- **Finding the Contact Set**. Illustrated with two moving triangles in 3D. How the triangles are just touching depends on which vertices project to which extremes of the projection interval: All 3 project to same point, 2 project to min and 1 projects to max, 1 projects to min and 2 project to max, 1 projects to min and 1 projects to max.

```
ProjectionMap { M3, M21, M12, M11 };
Config { ProjectionMap map; int index[3]; float min, max; };
Config GetConfiguration (Point D, Point U[3])
{   d0 = Dot(D,U[0]);  d1 = Dot(D,U[1]);  d2 = Dot(D,U[2]);
    if ( d0 <= d1 )
    {   if ( d1 <= d2 )  // d0 <= d1 <= d2
        {   cfg.map = ((d0!=d1)?(d1!=d2?M11:M12):(d1!=d2?M21:M3));
            cfg.index[] = { 0,1,2 };
            cfg.min = d0;  cfg.max = d2;
        }
        else if ( d0 <= d2 )  // d0 <= d2 < d1
        {   if ( d0 != d2 )
            {   cfg.map = M11;
                cfg.index[] = { 0,2,1 };
            }
            else
            {   cfg.map = M21;
                cfg.index[] = { 2,0,1 };  // keep even permutation!
            }
            cfg.min = d0;  cfg.max = d1;
        }
        else  // d2 < d0 <= d1
        {   cfg.map = ( d0 != d1 ? M11 : M12 );
            cfg.index[] = { 2,0,1 };
            cfg.min = d2;  cfg.max = d1;
        }
    }
    else if ( d2 <= d1 )  // d2 <= d1 < d0
    {   if ( d2 != d1 )
        {   cfg.map = M11;
            cfg.index[0] = { 2,1,0 };
        }
        else
        {   cfg.map = M21;
            cfg.index[] = { 1,2,0 };  // keep even permutation!
        }
        cfg.min = d2;  cfg.max = d0;
    }
    else if ( d2 <= d0 )  // d1 < d2 <= d0
    {   cfg.map = ( d2 != d0 ? M11 : M12 );
        cfg.index[] = { 1,2,0 };
        cfg.min = d1;  cfg.max = d0;
    }
    else  // d1 < d0 < d2
    {   cfg.map = M11;
        cfg.index[] = { 1,0,2 };
        cfg.min = d1;  cfg.max = d2;
    }
}
```

```
bool Update (Config UC, Config VC, float speed,
    Side& side, Config& TUC, Config& TVC, float& TFirst, float& TLast)
{
    if ( VC.max < UC.min )  // V-interval initially on ‘left’ of U-interval
    {
        if ( speed <= 0 ) return false;  // intervals moving apart
        T = (UC.min - VC.max)/speed;
        if ( T > TFirst ) { TFirst = T; side = LEFT; TUC = UC; TVC = VC; }
        T = (UC.max - VC.min)/speed;  if ( T < TLast ) TLast = T;
        if ( TFirst > TLast ) return false;
    }
    else if ( UC.max < VC.min )  // V-interval initially on ‘right’ of U-interval
    {
        if ( speed >= 0 ) return false;  // intervals moving apart
        T = (UC.max - VC.min)/speed;
        if ( T > TFirst ) { TFirst = T; side = RIGHT; TUC = UC; TVC = VC; }
        T=  (UC.min - VC.max)/speed;  if ( T < TLast ) TLast = T;
        if ( TFirst > TLast ) return false;
    }
    else  // U-interval and V-interval overlap
    {
        if ( speed > 0 )
        {
            T = (UC.max - VC.min)/speed;
            if ( T < TLast ) TLast = T;  if ( TFirst > TLast ) return false;
        }
        else if ( speed < 0 )
        {
            T = (UC.min - VC.max)/speed;
            if ( T < TLast ) TLast = T;  if ( TFirst > TLast ) return false;
        }
    }
    return true;
}
```

```
ContactSet GetFirstContact (Point U[3], Point W0, Point V[3], Point W1,
    Side side, Config TUC, Config TVC, float TFirst)
{
    // move triangles to first contact
    Point UTri[3] = { U[0]+TFirst*W0, U[1]+TFirst*W0, U[2]+TFirst*W0 };
    Point VTri[3] = { V[0]+TFirst*W1, V[1]+TFirst*W1, V[2]+TFirst*W1 };
    Segment USeg, VSeg;
    if ( side == RIGHT )  // V-interval on right of U-interval
    {   if ( TUC.map == M21 || TUC.map == M111 ) return UTri[TUC.index[2]];
        if ( TVC.map == M12 || TVC.map == M111 ) return VTri[TVC.index[0]];
        if ( TUC.map == M12 )
        {   USeg = <UTri[TUC.index[1]],UTri[TUC.index[2]]>;
            if ( TVC.map == M21 )
            {   VSeg = <VTri[TVC.index[0]],VTri[TVC.index[1]]>;
                return SegSegIntersection(USeg,VSeg); }
            else  // TVC.map == M3
            {   return SegTriIntersection(USeg,VTri); }
        }
        else  // TUC.map == M3
        {   if ( TVC.map == M21 )
            {   VSeg = <VTri[TVC.index[0]],VTri[TVC.index[1]]>;
                return SegTriIntersection(VSeg,UTri); }
            else  // TVC.map == M3
            {   return CoplanarTriTriIntersection(UTri,VTri); }
        }
    }
    else if ( side == LEFT )  // V-interval on left of U-interval
    {   if ( TVC.map == M21 || TVC.map == M111 ) return VTri[TVC.index[2]];
        if ( TUC.map == M12 || TUC.map == M111 ) return UTri[TUC.index[0]];
        if ( TVC.map == M12 )
        {   VSeg = <VTri[TVC.index[1]],VTri[TVC.index[2]]>;
            if ( TUC.map == M21 )
            {   USeg = <UTri[TUC.index[0]],UTri[TUC.index[1]]>;
                return SegSegIntersection(USeg,VSeg); }
            else  // TUC.map == M3
            {   return SegTriIntersection(VSeg,UTri); }
        }
        else  // TVC.map == M3
        {   if ( TUC.map == M21 )
            {   USeg = <UTri[TUC.index[0]],UTri[TUC.index[1]]>;
                return SegTriIntersection(USeg,VTri); }
            else  // TUC.map == M3
            {   return CoplanarTriTriIntersection(UTri,VTri); }
        }
    }
    else  // triangles were initially intersecting
    {   return TriTriIntersection(UTri,VTri); }
}
```

```
bool TrianglesIntersect (Point U[3], Point W0, Point V[3], Point W1,
    float& TFirst, float& TLast, ContactSet& contact)
{
    W = W1 - W0;
    S = set of all potential separating axes;
    TFirst = 0;  TLast = INFINITY;
    side = NONE;
    Config TUC, TVC;

    for each D in S do
    {
        speed = Dot(D,W);
        Config UC = GetConfiguration(D,U);
        Config VC = GetConfiguration(D,V);
        if ( !Update(UC,VC,speed,side,TUC,TVC,TFirst,TLast) )
            return false;
    }

    contact = GetFirstContact(U,W0,V,W1,side,TUC,TVC,TFirst);
    return true;
}
```

Section 12. Curves.

- Definitions and basic concepts (7.1, 7.2, 7.3)

- **Curves.** Useful in games for scripted paths (travel along path with constant speed, change orientation based on how sharp an object turns along path). Also useful for tube surfaces and blended surfaces from curve boundaries.

- **Definitions.** A parametric curve in n–dimensions is $\vec{X}(t)$ where $t \in [a, b]$ (typically $[0, 1]$).
  - End points: $\vec{X}(a)$, $\vec{X}(b)$
  - Tangent (velocity): $\vec{X}'(t)$, the derivative with respect to t
  - Speed: $|\vec{X}'(t)|$
  - Arc length: $s(t) = \int_a^t |\vec{X}'(\tau)| d\tau$
  - Total length: $L = s(b)$
  - Normalized arc length: $\eta(t) = s(t)/L \in [0, 1]$
  - Curve $\vec{X}(s)$ is parameterized by arc length s if $|\vec{X}'(s)| \equiv 1$.

- **Coordinate Frame for Planar Curve.** If the tangent vector is $\vec{T}(s) = (\cos(\theta(s)), \sin(\theta(s)))$, then a normal vector is $\vec{N}(s) = (-\sin(\theta(s)), \cos(\theta(s)))$. The vectors form a coordinate frame satisfying

$$\begin{aligned} \vec{T}'(s) &= \kappa(s)\vec{N}(s) \\ \vec{N}'(s) &= -\kappa(s)\vec{T}(s) \end{aligned}$$

  where $\kappa(s) = \theta'(s)$ is the curvature of the curve. In terms of $(x(t), y(t))$,

$$\kappa(t) = \frac{x'(t)y''(t) - x''(t)y'(t)}{((x'(t))^2 + (y'(t))^2)^{3/2}}.$$

- **Coordinate Frame for Space Curve.** Tangent is $\vec{T}(s)$, normal is $\vec{N}(s)$, binormal is $\vec{B}(s) = \vec{T}(s) \times \vec{N}(s)$. The vectors form a coordinate frame satisfying

$$
\begin{aligned}
\vec{T}'(s) &= \kappa(s)\vec{N}(s) \\
\vec{N}'(s) &= -\kappa(s)\vec{T}(s) + \tau(s)\vec{B}(s) \\
\vec{B}'(s) &= -\tau(s)\vec{N}(s)
\end{aligned}
$$

  where $\kappa(s)$ is curvature and $\tau(s)$ is torsion. In terms of t,

$$\kappa(t) = \pm\frac{|\vec{X}' \times \vec{X}''|}{|\vec{X}'|^3}$$

  and

$$\tau(t) = \frac{\vec{X}' \cdot \vec{X}'' \times \vec{X}'''}{|\vec{X}' \times \vec{X}''|^2}.$$

- **Reparameterize by Arc Length.** Standard question is how to select points that are equally spaced along curve. The idea is to move an object (perhaps the camera) along a path with constant speed. Given uniform samples in time, compute positions that are uniformly spaced in distance (measured as arc length along curve). That is, select $t_i \in [a, b]$ for $0 \le i \le n$ with $t_0 = a$ and $t_n = b$. If $s_i = s(t_i)$ are the arc lengths, we want $s_i = Li/n$.

  The basic problem is to select s and solve for t in the integral for arc length. In most cases, the integral is not expressible in terms of elementary functions. Even if it were, we need to invert the result. If $s = f(t)$, we want $t = f^{-1}(s)$. Inversion itself is usually not possible in terms of elementary functions. Must resort to numerical methods.

  Define $F(t) = f(t) - s$ for $t \in [a, b]$ and for a specified s. We want a t for which $F(t) = 0$. Use Newton's method. Select initial guess T_0. Iterate

$$T_{i+1} = T_i - \frac{F(T_i)}{F'(T_i)}, \quad i \ge 0$$

  until convergence criteria are met ($F(T_i)$ close enough to zero, $T_{i+1} - T_i$ is close enough to zero, too many iterations, etc.).

```
Input:   tmin, tmax, L, s in [0,L], and a curve X(t)
Output:  t in [tmin,tmax] corresponding to s

// Choose initial guess based on relative location
// of s in [0,L]
ratio = s/L;
t = (1-ratio)*tmin + ratio*tmax;

// iterate using Newton's method
for (i = 0; i < IMAX; i+)
{
    F = X.Length(t) - s;
    if ( |F| < EPSILON )  // might also track/test |t1-t0|
        return t;

    DF = X.Speed(t);
    t = t - F/DF;  // warning:  What if DF nearly zero?
}

// No convergence, return best guess?  Inform caller?
return t;
```

- **Degree Reduction**. An attempt to approximate a given polynomial curve $\vec{X}(t)$ by another polynomial curve $\vec{Y}(t)$ such that degree($\vec{Y}$) $<$ degree($\vec{X}$) and norm($\vec{X}, \vec{Y}$) $< \varepsilon$. For simplicity, assume the domain is $[0, 1]$.

  A nonstandard approach from a computer graphics perspective is to use a least squares approximation using the L^2 norm, possibly with constraints. If $\vec{X}(t) = \sum_{i=0}^{n} \vec{a}_i t^i$, $\vec{Y}(t) = \sum_{j=0}^{m} \vec{b}_j t^j$, $m < n$, the unknowns $\vec{b}_j$ are determined by minimizing

  $$E(\vec{b}_0, \ldots, \vec{b}_m) = \int_0^1 |\vec{X}(t) - \vec{Y}(t)|^2 \, dt.$$

  This reduces to solving a linear system of equations. A variation with constraints is to force end point equality, $\vec{X}(0) = \vec{Y}(0)$ and $\vec{X}(1) = \vec{Y}(1)$. Function E has two less vector components in this case.

Section 13. Curves.

- Subdivision (7.4)

- **Subdivision.** Given a curve $\vec{X}(t)$ for $t \in [a,b]$, a subdivision is a set of points $\vec{X}_i = \vec{X}(t_i)$ where $t_i \in [a,b]$ for $0 \leq i \leq n$ and $t_{i+1} > t_i$. The implied polyline is an approximation to the curve.
- **Subdivision by Uniform Sampling.** Select $t_i = a + (b-a)i/n$. Easy to compute points, but polyline may not be a good approximation to the curve. Consider a worst case example of $\vec{X}(t) = (t, \sin(t))$ for $t \in [0, 2\pi]$; $t_0 = 0$, $t_1 = \pi$, $t_2 = 2\pi$. (Problem is Nyquist frequency. n not large enough to capture high frequencies in curve.)
- **Subdivision by Arc Length.** Total length of curve is L. Compute t_i corresponding to arc length $s_i = Li/n$. Need to use the algorithm for reparameterization by arc length. Useful for sampling a curve that represents the path of an object/camera that is traveling at constant speed. This method can also miss variation in the curve.
- **Subdivision by Midpoint Distance.** Recursively bisect $[a,b]$. The bisection step on an interval $[t_\ell, t_r]$ is performed if the distance between $\vec{X}((t_\ell + t_r)/2)$ and the line segment connecting $\vec{X}(t_\ell)$ and $\vec{X}(t_r)$ is smaller than a prescribed tolerance. Beware of pathological cases such as $\vec{X}(t) = (t, \sin(t))$ for $t \in [0, 2\pi]$. Similar pathologies if bisection is based on $\vec{X}''$ at the midpoint.

```
void Bisect (int level, float t0, Point x0, float t1,
    Point x1, List L)
{   if ( level > 0 )  // control maximum recursion depth
    {   tm = (t0+t1)/2;
        xm = x(tm);
        d0 = length of segment <x0,x1>;
        d1 = distance from xm to segment <x0,x1>;
        if ( d1/d0 > epsilon )
        {   Bisect(level-1,t0,x0,tm,xm);
            Bisect(level-1,tm,xm,t1,x1);
            return;
        }
    }
    L.AddToEnd(x1);
}
```

```
// x(t) is curve on [tmin,tmax]
maxLevel = <user specified>;
L = { x(tmin) };
Bisect(maxLevel,tmin,x(tmin),tmax,x(tmax),L);
```

- **Subdivision by Variation.** A global approach to error control. Recursively bisect if the variation between the curve on the subinterval and the line segment connecting the end points of the interval is small enough. For subinterval $[t_\ell, t_r]$ with $t_m = (t_\ell + t_r)/2$, variation is

$$V = \int_{t_\ell}^{t_r} |\vec{X}(t) - \vec{L}(t)|^2 dt$$

  where $\vec{L}(t)$ is the line segment connecting $\vec{X}(t_\ell)$ and $\vec{X}(t_r)$.

- **Subdivision by Minimizing Variation.** Select n ahead of time (a "point budget"). Define V_i to be the variation on $[t_i, t_{i+1}]$ between the curve and the line segment connecting the end points. Define

$$E(t_0, ..., t_n) = \sum_{i=0}^{n} V_i.$$

  Select the t_i that minimizes E.

- **Subdivision by Curvature.** A couple of ideas. One is to compute total absolute curvature K (integral of absolute curvature over parameter domain). Uniformly subdivide $[0, K]$ into $\kappa_i = Ki/n$. Determine t_i for which the integral of absolute curvature on $[0, t_i]$ is κ_i. Another idea is to select n ahead of time. Treat the points $\vec{X}(t_i)$ as positively charged particles that repel each other. Instead of using inverse squared distance law, replace distance by absolute curvature.

- **Fast Subdivision of Cubic Curves.** Use recursive bisection based on second derivative of curve at midpoint. The curve can be represented as a Taylor polynomial (expanded at t) by

$$\vec{X}(t \pm d) = \vec{X}(t) \pm d\vec{X}'(t) + \frac{d^2}{2}\vec{X}''(t) \pm \frac{d^3}{6}\vec{X}'''(t).$$

Leads to the two identities

$$\vec{X}(t) = [\vec{X}(t+d) + \vec{X}(t-d) - d^2\vec{X}''(t)]/2$$

and

$$\vec{X}''(t) = [\vec{X}''(t+d) + \vec{X}''(t-d)]/2.$$

If you know $\vec{X}(t \pm d)$ and $\vec{X}''(t \pm d)$, then you can compute $\vec{X}(t)$ and $\vec{X}''(t)$.

```
void Subdivide (float t0, float t1, Point x0, Point x1,
    Point sd0, Point sd1, List L)
{
    sdmid = (sd0 + sd1)/2;
    d = t1 - t0;
    dsqr = d*d;
    nonlinearity = dsqr*sdmid;
    if ( Length(nonlinearity) > epsilon )
    {
        tmid = (t0 + t1)/2;
        xmid = (x0 + x1 - nonlinearity)/2;
        L.InsertBetween(xmid,x0,x1);
        Subdivide(t0,tmid,x0,xmid,sd0,sdmid);
        Subdivide(tmid,t1,xmid,x1,sdmid,sd1);
    }
}

x0 = x(tmin);
x1 = x(tmax);
sd0 = x"(tmin);
sd1 = x"(tmax);
L = { x0, x1 };
Subdivide(tmin,tmax,x0,x1,sd0,sd1,L);
```

Section 14. Surfaces.

- Definitions and basic concepts (8.1, 8.2, 8.3)
- Subdivision (8.4)

- **Surfaces.** Useful in games for building smoother looking models. Polygon models have a fixed level of detail (you can always reduce the level of detail). Surfaces have "infinite level of detail", but you do choose a level (perhaps dynamically) and tessellate the surfaces to generate triangles for the renderer. Good for game consoles (limited memory, lots of processing power).

- **Definitions.** A parametric rectangle patch is a function $\vec{X}(s,t)$ where $(s,t) \in [s_0, t_1] \times [s_0, t_1]$ (typically $[0,1] \times [0,1]$).

  - Corner points: $\vec{X}(s_i, t_j)$ (4 choices)
  - Boundary curves: $\vec{X}(s_i, t)$, $\vec{X}(s, t_j)$ (4 choices)

  A parametric triangle patch has (s,t) in a triangle domain, typically $s \geq 0$, $t \geq 0$, and $s + t \leq 1$.

  - Corner points: $\vec{X}(0,0)$, $\vec{X}(1,0)$, $\vec{X}(0,1)$
  - Boundary curves: $\vec{X}(s,0)$, $\vec{X}(0,t)$, $\vec{X}(s,s)$

  In both cases,

  - Tangent basis: $\vec{X}_s(s,t)$, $\vec{X}_t(s,t)$ (subscripts indicate partial derivatives)
  - Normal field: $\vec{X}_s \times \vec{X}_t$ (normalize for unit length vectors)
  - Surface area: $\int\int_{D_0} |\vec{X}_s \times \vec{X}_t| \, ds \, dt$ for D_0 a subset of domain D for (s,t).
  - Surface curvature: The stuff of differential geometry. Heavy on the math.

- **Degree Reduction.** An attempt to approximate a given polynomial patch $\vec{X}(s,t)$ by another polynomial patch $\vec{Y}(s,t)$ such that degree($\vec{Y}$) < degree($\vec{X}$) and norm($\vec{X}, \vec{Y}$) $< \varepsilon$. Same nonstandard approach as for curves, use a least squares approach and minimize

  $$E(\vec{\rho}) = \int_D |\vec{X}(s,t) - \vec{Y}(s,t;\vec{\rho})|^2 \, ds \, dt.$$

  A bit more challenging than for curves. You need to handle reduction on shared boundaries in a way to guarantee at least continuity across the boundary.

- **Subdivision of surfaces.** Many ways to do this, just like for curves. Uniform subdivision is "easy" to implement, but recursive subdivision is faster. Illustrated for bicubic Bézier patches,

$$\vec{X}(s,t) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_{3,i}(s)B_{3,j}(t)\vec{P}_{i,j}$$

  where $B_{n,i}(r)$ is a Bernstein polynomial. Do an operation count for a single evaluation. Get about 90 operations.

```
void UniformSubdivide (int level, BezierPatch X, int& size,
    Point vertex[size][size])
{
    p = pow(2,level);
    size = p+1;
    allocate vertex[][];
    for (i = 0; i < size; i++)
    {
        s = i/p;
        for (j = 0; j < size; j++)
        {
            t = j/p;
            vertex[i][j] = X(s,t);  // evaluate patch
        }
    }
}
```

- **Fast subdivision.** Similar to curves. Uses the identities

$$\begin{aligned}
\vec{X}(s,t) &= \left(\vec{X}(s+d,t) + \vec{X}(s-d,t) - d^2\vec{X}_{ss}(s,t)\right)/2 \\
\vec{X}(s,t) &= \left(\vec{X}(s,t+d) + \vec{X}(s,t-d) - d^2\vec{X}_{tt}(s,t)\right)/2 \\
\vec{X}_{ss}(s,t) &= \left(\vec{X}_{ss}(s+d,t) + \vec{X}_{ss}(s-d,t)\right)/2 \\
\vec{X}_{tt}(s,t) &= \left(\vec{X}_{tt}(s,t+d) + \vec{X}_{tt}(s,t-d)\right)/2
\end{aligned}$$

If rectangle subdomain is $[s_0, s_1] \times [t_0, t_1]$ and if s_m and t_m are midpoints with $d = s_m - s_0 = t_m - t_0$, the idea is to compute $\vec{X}_{ss}(s_m, \bullet)$, $\vec{X}_{sstt}(s_m, \bullet)$, $\vec{X}_{tt}(s_m, \bullet)$, $\vec{X}(s_m, \bullet)$. Similar construction for points $(\bullet, t_m)$. Finally, compute center $\vec{X}(s_m, t_m)$. The operation count is a bit tedious, but is about 34 operations per vertex in the subdivision (about `2.5` times faster than the double loop evaluation). Some issues.

– The most obvious implementation of recursive subdivision will compute interior vertices multiple times (at most twice).

– If patch is flat in a subregion, why subdivide? Want nonuniform subdivision. Need to worry about cracking.

– If patch is far from camera, why subdivide? Want the subdivision to be controlled both by geometry of patch and by screen space metric.

Section 15. Animation.

- Basic concepts (9)
- Key frame animation (9.1)
- TCB splines (7.3.4)

- **Animation.** The process of controlling any time–varying quantity in a scene graph. The implementation in the book uses the concept of a *controller* attached to a node that modifies the behavior of that node (examples: modifies transforms, model vertex data, render state). During the `UpdateGS` pass, the `UpdateWorldData` method is called and gives each controller a chance to modified the data it controls.

  Special topics to be discussed: key frame animation, inverse kinematics, skin–and–bones, morphing.

- **Key Frame Animation.** At a single node in the scene graph, a sequence of transforms with corresponding times is provided, each transform representing a *pose* of the represented object at its corresponding time. Each time–transform pair is called a *key frame* or simply *key*. The animation is obtained by computing the *between* transforms via some type of interpolation.

  In the book implementation, transforms are separated into rotation, translation, and scale. A key frame system allows sequences of each type of transform at a node. It is not required that multiple sequences at a node all have the same number of keys.

  Many forms of interpolation can be applied, but in most cases the interpolator is required to be exact; that is, the original keys must lie on the interpolated curve (preserve the artist's data). Most popular:

  – **Translation**. Use TCB splines (Kochanek, Bartels). T is tension which controls how sharply the curve bends at a control point. C is continuity which controls the visual variation in continuity at a control point, B is bias which controls the direction of the path at a control point by computing a weighted combination of one–sided derivatives at the control point.

  – **Rotation**. Use quaternions to represent the rotation matrices. Use SLERP (spherical linear interpolation) or SQUAD (spherical quadratic interpolation).

  – **Scale**. Given two scales s_0 and s_1 and a time $t \in [0,1]$, either use a weighted arithmetic mean $(1-t)s_0 + ts_1$ or a weighted geometric mean $s_0^{(1-t)} s_1^t$.

- **TCB Splines.** Given consecutive positions $\vec{P}_i$ and $\vec{P}_{i+1}$ and tangent vectors $\vec{T}_i$ and $\vec{T}_{i+1}$, use a Hermite interpolation basis $H_0(t) = 2t^3 - 3t^2 + 1$, $H_1(t) = -2t^3 + 3t^2$, $H_2(t) = t^3 - 2t^2 + t$, and $H_3(t) = t^3 - t^2$ to create a curve
$$\vec{X}_i(t) = H_0(t)\vec{P}_i + H_1(t)\vec{P}_{i+1} + H_2(t)\vec{T}_i + H_3(t)\vec{T}_{i+1}$$
for $t \in [0, 1]$. Catmull–Rom splines is special case where $\vec{T}_i = (\vec{P}_{i+1} - \vec{P}_{i-1})/2$.

  Allow outgoing tangent at $t = 0$, $\vec{T}_i^0$. Allow incoming tangent at $t = 1$, $\vec{T}_{i+1}^1$. Use incoming tangent instead of $\vec{T}_i$ and outgoing tangent instead of $\vec{T}_{i+1}$ in curve formula.

  - Introduce tension parameter $\tau \in [0, 1]$,
  $$\vec{T}_i^0 = \vec{T}_i^1 = \frac{1-\tau}{2}\left(\left(\vec{P}_{i+1} - \vec{P}_i\right) + \left(\vec{P}_i - \vec{P}_{i-1}\right)\right).$$
  Catmull–Rom is $\tau = 0$. Increase τ tightens the curve at the control point. Decrease τ slackens the curve.
  - Introduce continuity parameter $\gamma \in [-1, 1]$,
  $$\vec{T}_i^0 = \left(\frac{1-\gamma}{2}\left(\vec{P}_{i+1} - \vec{P}_i\right) + \frac{1+\gamma}{2}\left(\vec{P}_i - \vec{P}_{i-1}\right)\right)$$
  and
  $$\vec{T}_i^1 = \left(\frac{1+\gamma}{2}\left(\vec{P}_{i+1} - \vec{P}_i\right) + \frac{1-\gamma}{2}\left(\vec{P}_i - \vec{P}_{i-1}\right)\right).$$
  Curve tangent is continuous when $\gamma = 0$. Increase γ to produce a corner in the curve. Direction of corner depends on sign of γ.
  - Introduce bias parameter $\beta \in [-1, 1]$,
  $$\vec{T}_i^0 = \vec{T}_i^1 = \left(\frac{1-\beta}{2}\left(\vec{P}_{i+1} - \vec{P}_i\right) + \frac{1+\beta}{2}\left(\vec{P}_i - \vec{P}_{i-1}\right)\right).$$
  Equally weighted one–sided tangents when $\beta = 0$. For β near -1, outgoing tangent dominates the direction of the path. For β near 1, incoming tangent dominates the direction of the path.

– Combine all three parameters

$$\begin{aligned}
\vec{T}_i^0 &= \left(\tfrac{(1-\tau)(1-\gamma)(1-\beta)}{2}\left(\vec{P}_{i+1}-\vec{P}_i\right)\right. \\
&\quad \left. + \tfrac{(1-\tau)(1+\gamma)(1+\beta)}{2}\left(\vec{P}_i-\vec{P}_{i-1}\right)\right)
\end{aligned}$$

and

$$\begin{aligned}
\vec{T}_i^1 &= \left(\tfrac{(1-\tau)(1+\gamma)(1-\beta)}{2}\left(\vec{P}_{i+1}-\vec{P}_i\right)\right. \\
&\quad \left. + \tfrac{(1-\tau)(1-\gamma)(1+\beta)}{2}\left(\vec{P}_i-\vec{P}_{i-1}\right)\right).
\end{aligned}$$

Section 16. Animation.

- Quaternions (2.3)
- Representations of rotations (space/time trade offs)
- Interpolation of rotations (9.1.4)

- **Quaternions.** Given a rotation about an axis $\vec{U} = (u_0, u_1, u_2)$ by angle θ, the quaternion representing the rotation is $q = \cos(\theta/2) + \hat{U}\sin(\theta/2)$ where $\hat{U} = u_0 i + u_1 j + u_2 k$ (proof is detailed).

  Game engine libraries typically have conversions between rotation matrices, axis–angle form, and quaternions. Which form is 'better'? Another set of trade offs. Memory: Rotation matrices use 9 float. Quaternions and angle–axis each use 4 floats.

- **Transforming.**
  - **Matrix form.** $R\vec{V}$ uses $9m + 6a$. Total = 15 ops.
  - **Angle–axis form.**

    $$R\vec{V} = \vec{V} + (\sin\theta)\vec{U} \times \vec{V} + (1 - \cos\theta)\vec{U} \times (\vec{U} \times \vec{V}).$$

    Need to compute $\sin(\theta)$ and $\cos(\theta)$. Using math libs, can be very expensive. Can use polynomial approximations or tables instead.

    Precompute $\sin\theta$ and $1 - \cos\theta$. Increase memory usage to 6 floats (still cheaper than matrices). $\vec{U} \times \vec{V}$ uses $6m + 3a$. $\vec{U} \times (\vec{U} \times \vec{V})$ uses $6m + 3a$. Multiply each of the cross products by a scalar, $6m$. Add three vectors, $6a$. Total = $18m + 12a$ = 30 ops.
  - **Quaternion form.** $R\vec{V}$ computed as the imaginary part of $q\hat{V}q^*$. Product of two quaternions uses $16m + 12a$. But $\hat{V}$ has no real part, so $q\hat{V}$ takes $12m+8a$. Product $(q\hat{V})q^*$ has no real part, uses $12m+9a$. Total = $24m + 17a$ = 41 ops.

    However, conversion from quaternion to matrix uses $12m + 12a$ operations. Matrix form uses 15 ops, so Total = 39 ops.

    For transforming N vectors, matrix form uses $15N$ operations, angle–axis form uses $30N$ operations, quaternion form uses $24 + 15N$ operations. Unfortunately, hardware T & L cards require matrices, not quaternions.

- **Interpolation.** Neither the rotation form nor angle–axis form lead to a natural form of interpolation. Quaternions do. Need to interpolate points on the unit hypersphere in 4D.

- **Slerp.** Spherical linear interpolation. Motivation in 2D. Given two points p and q on the unit circle, what is the parameterization of the shortest arc between them, $f(t; p, q)$ for $t \in [0, 1]$, that has 'constant speed'. Turns out to be

  $$f(t) = \frac{\sin((1-t)\theta)p + \sin(t\theta)q}{\sin\theta}$$

  where θ is the angle between p and q. It can be shown that $|f'(t)| = |\theta/\sin\theta|$, a constant.

  Same formula works even if p and q are unit vectors in higher dimensions.

- **Squad.** Spherical quadratic interpolation. Higher–degree fit of points (really a cubic interpolation), get a smoother fit, but requires 3 Slerps. Can modify to support TCB style interpolation.

Section 17. Animation.

- Kinematics, forward and inverse (9.2)
- Skinning (9.3)

- **Kinematics.** The study of motion without considerations of mass or forces. Consider a polyline $\vec{P}_i$ for $0 \le i \le n$. Write as $\vec{P}_{i+1} = \vec{P}_i + L_i\vec{U}_i$. Last point $\vec{P}_n$ is a function of lengths L_i and unit directions $\vec{U}_i$ for $0 \le i < n$. The polyline is called a *manipulator* and $\vec{P}_n$ is called an *end effector*. Each $\vec{U}_i$ can be represented in a coordinate system with origin $\vec{P}_i$ and axis directions stored as the columns of a rotation matrix R_i. Given an Euler angle factorization of R_i, the corresponding angles are called the *joint angles* of the manipulator. So $\vec{U}_i = \vec{U}_i(\alpha_i, \beta_i, \gamma_i)$ and $\vec{P}_n = \vec{P}_n(\vec{L}, \vec{\alpha}, \vec{\beta}, \vec{\gamma})$.

- **Forward Kinematics.** Given L_i and $\vec{U}_i$, compute $\vec{P}_n$.

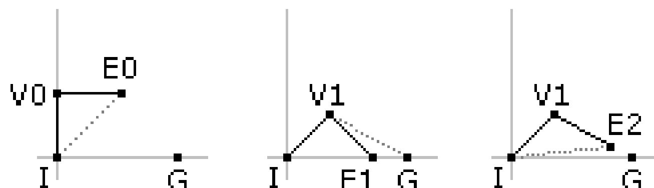
- **Inverse Kinematics.** Given $\vec{P}_n$, determine the L_i and $\vec{U}_i$ that yield the point if possible. If not, try to get "close enough". Given the desired *goal* $\vec{G}$, want to find a set of parameters to minimize $|\vec{P}_n(\vec{L}, \vec{\alpha}, \vec{\beta}, \vec{\gamma}) - \vec{G}|$. The problem is that there may be no solution or there may be multiple solutions (possibly infinitely many).

  In practice, the lengths $\vec{L}$ are fixed and the goal $\vec{G}$ varies continuously with time. Last set of joint angles of the manipulator is used as a starting point for computing the next set of joint angles (take what you get, constrain the parameters).

- **Variations.** Multiple goals (finite set of points, line, plane). Multiple end effectors. Tree manipulator. Physics parameters associated with manipulator (spring–like, elastic, damped motion).

- **Cyclic Coordinate Descent.** Method of solving the inverse kinematics problem. Minimize $|\vec{P}_n - \vec{G}|$ a joint at a time. Multiple passes made over the manipulator until some stopping criteria are met.

  Example. Initial point of manipulator is $\vec{I}$, end effector is $\vec{E}$, goal is $\vec{G}$. For unconstrained rotation, want $\vec{E}$ on line containing $\vec{I}$ and $\vec{G}$. For rotation constrained to a plane $\vec{N} \cdot (\vec{X} - \vec{I}) = 0$ (one degree of freedom), want $\vec{E}$ on line containing $\vec{I}$ and the projection of $\vec{G}$ onto the plane.



- **Skinning.** A *skin* is a triangle mesh whose world vertices are computed based on their relationship to a set of *bones* (transforms). Each bone has a list of vertices it affects. Each affected vertex has an offset $\vec{P}$ relative to the bone and a weight w. The incremental contribution to the final vertex is

  $$w\left(\sigma R\vec{P} + \vec{T}\right)$$

  where the bone has world scale σ, world rotation R, and world translation $\vec{T}$. Since the world vertices are computed for the skin, the skin and bones should share a common parent, but the world transform for the skin must be set to the identity (bones observe the world transform at the common parent, skin ignores it).

  Update is

  - Move the bones by changing their local transforms.
  - Update the scene graph to propagate transforms down the hierarchy. The bones are updated first, the skin mesh second.
  - When the skin mesh is visited, the world vertices are computed by iterating over all bones. For each bone, contributions are computed for the vertices it affects.
  - The skin mesh is sent to the renderer for drawing, but the renderer is told the world transform is the identity.

```
bool MgcSkinController::Update (MgcReal)
{
    // The skin vertices are calculated in the bone world
    // coordinate system.  The bone world coordinates already
    // includes the world transform of the common parent, so
    // the world transforms for the skin mesh should be the
    // identity.
    MgcGeometry* pkGeom = (MgcGeometry*) m_pkObject;
    pkGeom->SetWorldTransformToIdentity();

    // set all vertices to the zero vector
    MgcVector3* akVertex = pkGeom->Vertices();
    memset(akVertex,0,pkGeom->GetVertexQuantity()*sizeof(MgcVector3));

    // update dependent vertices for each bone
    for (unsigned int uiB = 0; uiB < m_uiBoneQuantity; uiB++)
    {   MgcNode* pkBone = m_apkBone[uiB];
        MgcMatrix3 kSRot = pkBone->WorldScale()*pkBone->WorldRotate();
        unsigned int uiVMax = m_auiSkinVertexQuantity[uiB];
        SkinVertex* pkSV = m_aakSkinVertex[uiB];
        for (unsigned int uiV = 0; uiV < uiVMax; uiV++, pkSV++)
        {   MgcVector3 kVTrn = kSRot*pkSV->m_kOffset +
                pkBone->WorldTranslate();
            akVertex[pkSV->m_uiIndex] += pkSV->m_fWeight*kVTrn;
        }
    }

    // update vertex normals if the skin has them
    if ( pkGeom->Normals() )  pkGeom->UpdateModelNormals();

    pkGeom->UpdateModelBound();

    // controller computes world transform
    return true;
}
```

Section 18. Level of Detail.

- Basic concepts (10, 10.1, 10.2)
- Line mesh decimation (not in book, use `PolylineReduction.pdf`)

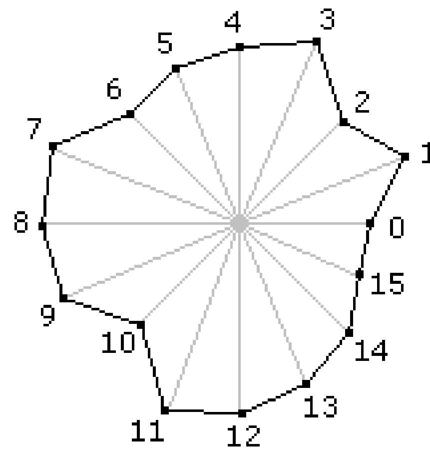
- **Level of Detail.** An object near the eye point require a large number of triangles to make it look realistic. The number of pixels covered by the rendered object is also large. If the object is far from the eye point, the number of pixels covered by the rendered object is small. The triangle–to–pixel ratio increases dramatically. No need to spend many cycles just to draw a few pixels.
  - **Sprites/Billboards.** 2D representations of 3D objects are used to reduce the complexity of the object. Example: Trees typically drawn as pair of rectangles with alpha blended textures, the pair intersecting in an 'X'. Example: Grandstands in a race. Audience typically drawn as rows of rectangular billboards that always face the camera.
  - **Discrete Level of Detail.** Create multiple representations of the same objecte. Use a switch node to select which one drawn. Selection based on distance of LOD center from eye point. Difference in triangle counts between consecutive models is typically large. Artists generate these, takes time.
  - **Continuous Level of Detail.** In the context of triangle decimation, really a discrete level of detail, but the difference in triangle count between models is small. Typically generated procedurally off line. Generating a good set of texture coordinates and normals can be difficult.
  - **Infinite Level of Detail.** Given a surface representation, tessellate as finely as you have the time. Typically generated at run time.
- **Line Mesh Decimation.** To illustrate the basic concepts for triangle mesh decimation, consider line meshs in the plane. Simplest example is an nonintersecting open polyline or a closed polyline that is a simple closed curve. Given three consecutive vertices $\vec{X}_{i-1}$, $\vec{X}_i$, $\vec{X}_{i+1}$, compute the distance from $\vec{X}_i$ to segment $\langle \vec{X}_{i-1}, \vec{X}_{i+1} \rangle$. If distance small compared to segment length, then the middle point is a good candidate for removal from the polyline. The ratio of distance to length is a *weight* assigned to the middle vertex. Compute weights for all points, remove the point of minimum weight. For an open polyline, assign infinite weight to the end points to keep it from shrinking. For a line mesh with points shared by three or more edges, assign such points infinite weight to preserve the topology.

- **Simple Algorithm.** Apply the reduction recursively. Compute the weights for all vertices. Remove the vertex of minimum weight to obtain a new polyline. Repeat the reduction on the new polyline. This is an $O(N^2)$ algorithm for N vertices.
- **Faster Algorithm.** When processing the reduced polyline, no need to recalculate weights at vertices that were unaffected by the reduction. In fact, if $\vec{X}_i$ is removed, only the weights change at $\vec{X}_{i-1}$ and $\vec{X}_{i+1}$. Can do in constant time. But you still need to search for the minimum weight, an $O(N)$ operation. You still have $O(N^2)$ for full reduction, but the constant in the asymptotics is smaller.
- **Even Better.** Use a min heap data structure that supports $O(1)$ lookup. Removal of minimum from heap requires $O(logN)$ update. Initial heap sorting takes $O(NlogN)$ time. When weights change, internal heap values change; heap can be updated in $O(logN)$ time when this happens (not a standard heap operation). If you have to search for the heap nodes corresponding to the changed weights, in worst case that is $O(N)$. However, you can store heap indices with the vertex information to support $O(1)$ lookup of the changed items. Full reduction is therefore $O(NlogN)$. For nonintersecting polylines (at most 2 edges per vertex), the heap node information is

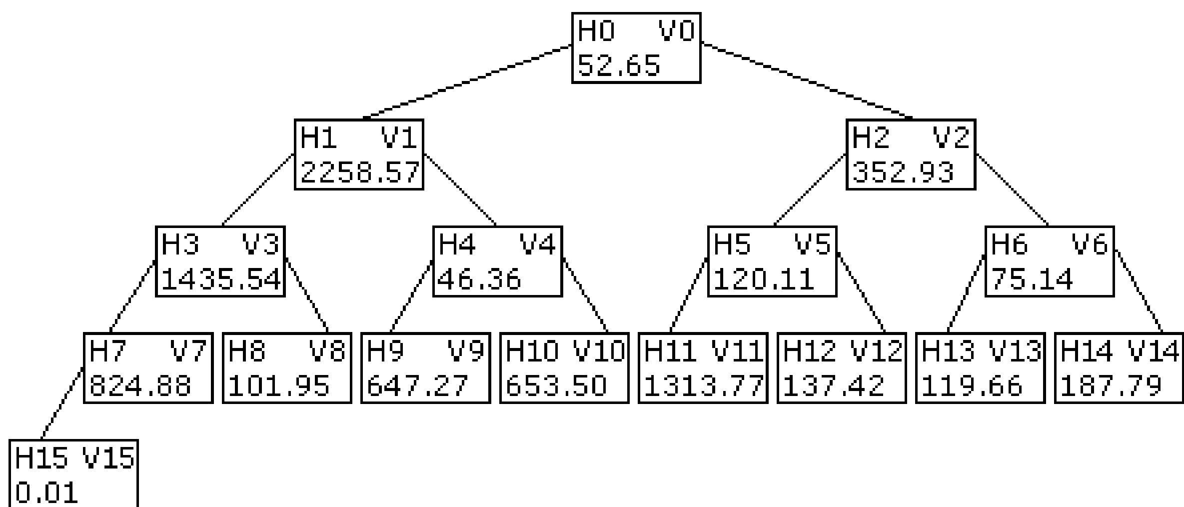
```
HeapRecord
{
    int V;          // vertex index
    int H;          // heap index
    float W;        // vertex weight
    HeapRecord* L;  // points to left vertex neighbor
    HeapRecord* R;  // points to right vertex neighbor
}
```

  Is more complicated with general line meshes.

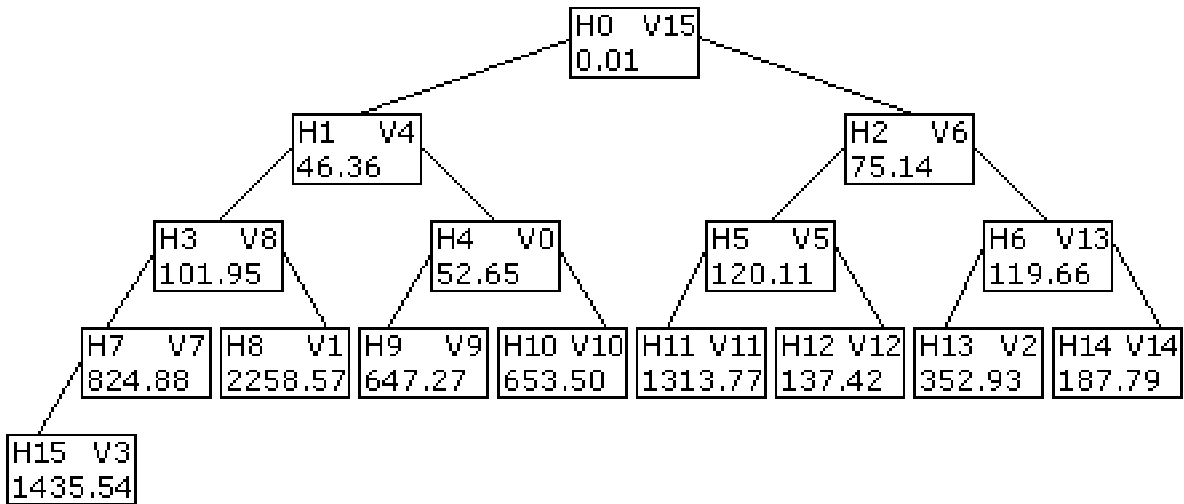
Example (see PolylineReduction.pdf file). Polygon is



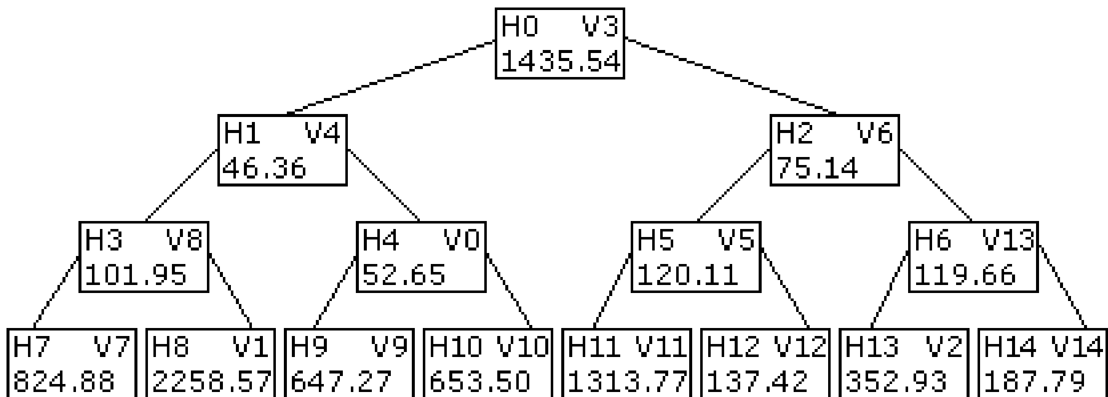
Initial heap binary tree is



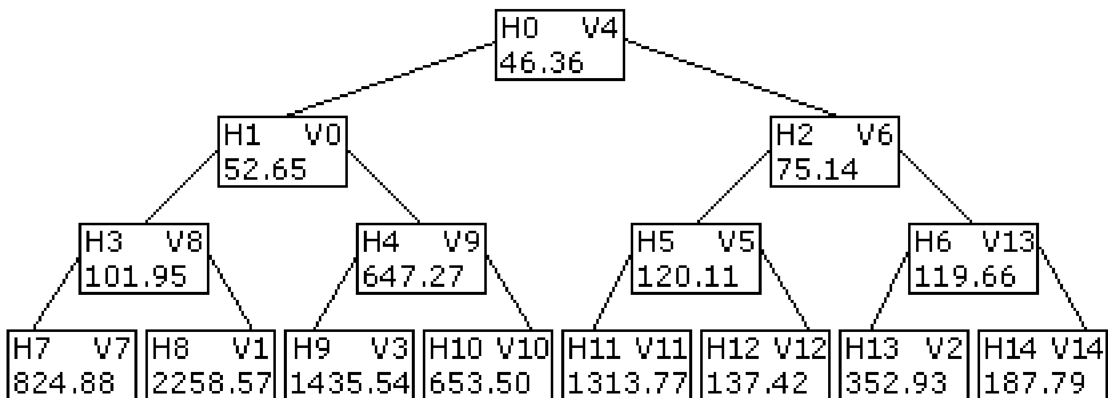
Sorted heap is



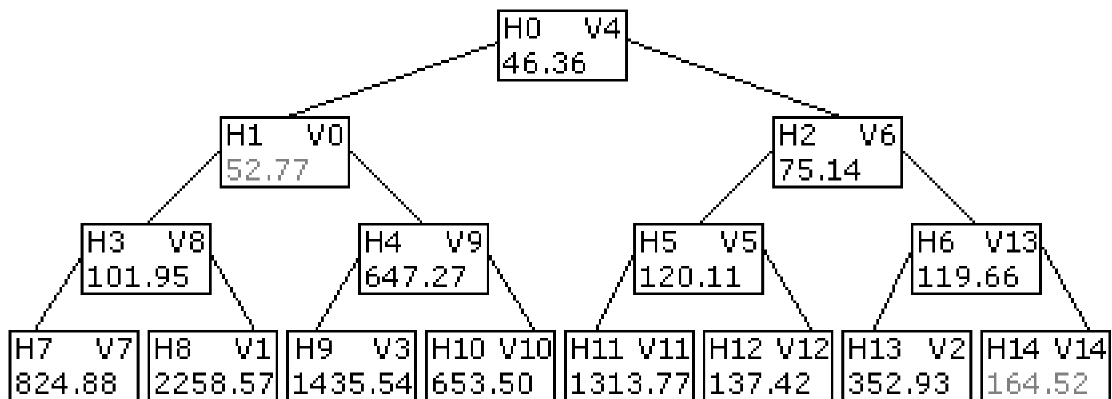
Remove minimum (vertex 15)



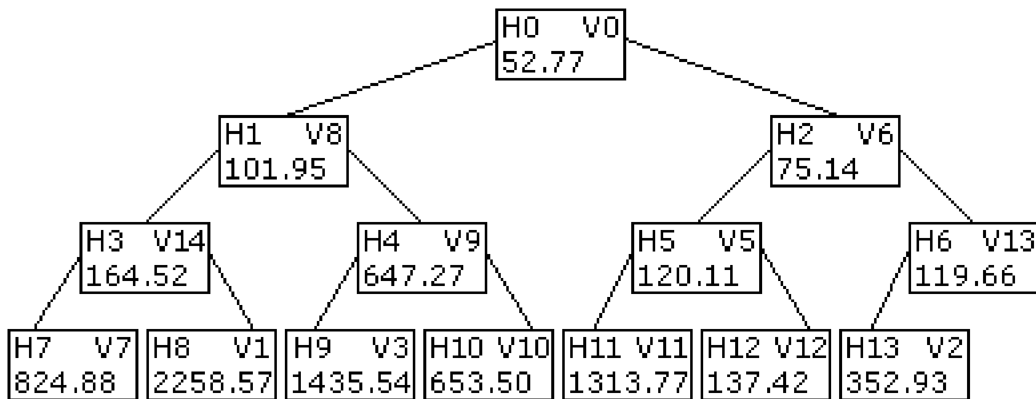
Update to restore to heap



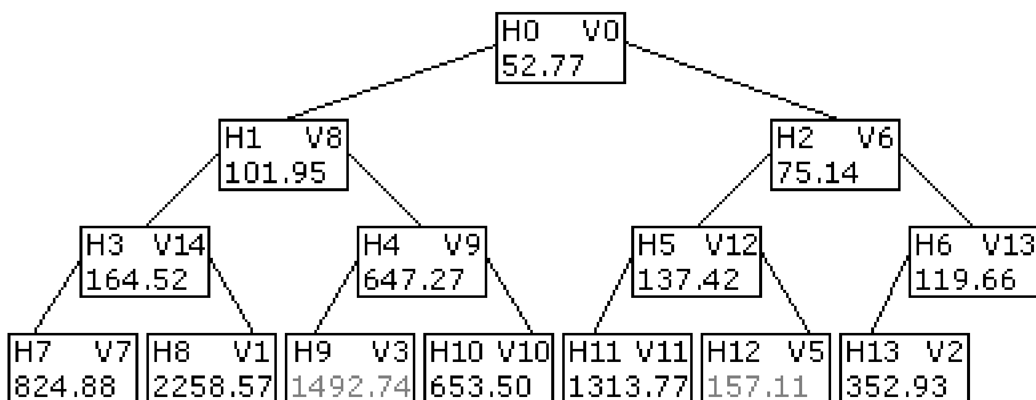
Weights at vertices 0 and 14 change (no update needed)



Remove minimum (vertex 4)



Weights at vertices 3 and 5 change (update needed at 5)

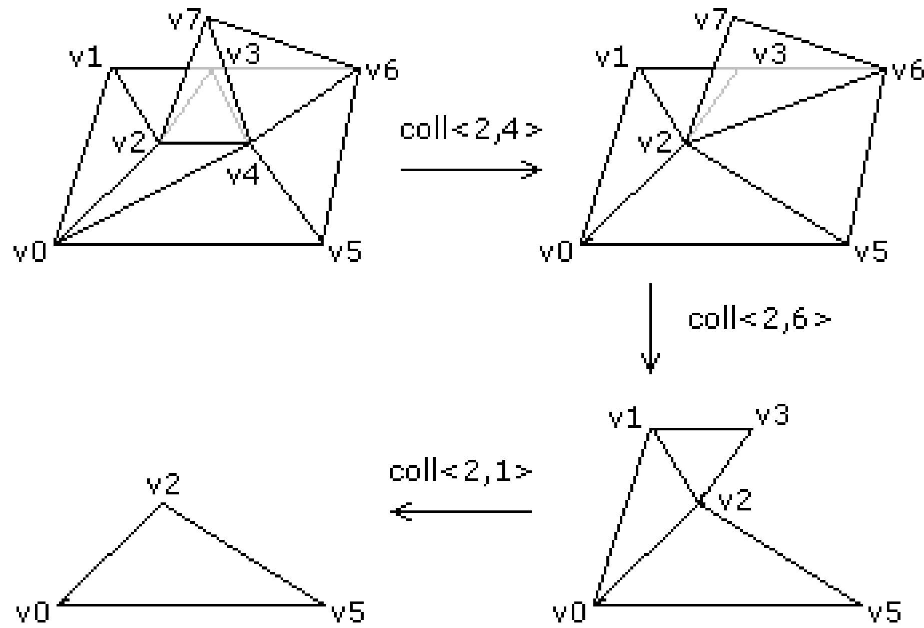


- **Dynamic change in LOD.** The reduction for a nonintersecting polyline generated a sequence of vertices to be removed. The heap record used a doubly–linked list for representing the polyline. At run time you can avoid the list handling and store edge connectivity in an array. For n vertices, array has $2n - 2$ entries grouped in pairs for open polyline, $2n - 1$ entries for closed polyline. Sort the index pairs so that the first edge removed is the last edge in the array.
  - **Decrease LOD.** Decrement edge–pair quantity by 1, replace appropriate index in first part of array by the removed vertex index.
  - **Increase LOD.** Increment edge–pair quantity by 1, restore appropriate index in first part of array. This requires remembering where you changed the index with each collapse. Can compute this mapping once (at decimation time), then just use during run time.
- **Vertex Reordering.** Permute the vertex array so that the first vertex removed is the last vertex in the array. Requires remapping the edge connectivity array. Supports batch transforming of contiguous blocks of vertices. Increase or decrease in LOD requires simply an increment or decrement of vertex quantity.

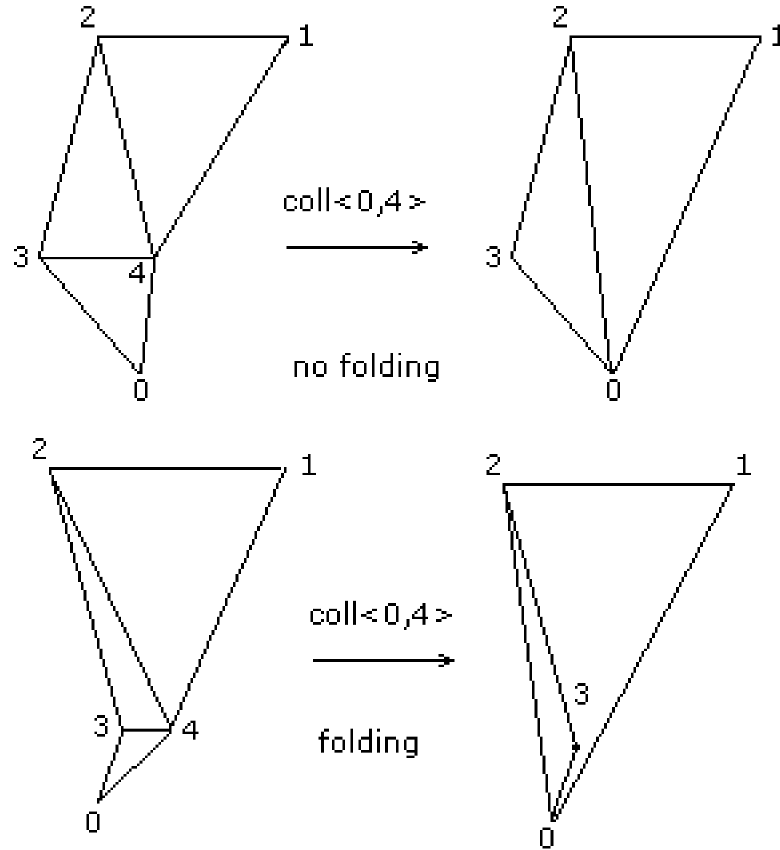
Section 19. Level of Detail.

- Triangle mesh decimation (10.3 with a lot of details added)

- **Triangle Mesh Decimation.** Same idea as for line meshes, but many more tedious details to take care of. Vertex collapse for a line mesh amounted to removing a vertex of minimum weight, then informing its right neighbor to connect itself to the left neighbor. For a triangle mesh, the concept is an *edge collapse*. An edge $\langle v_k, v_t \rangle$ of minimum weight is removed. The vertex v_k is the *keep* vertex and v_t is the *throw* vertex. The edge and v_t are removed from the mesh. All triangles sharing the edge are deleted. All remaining triangles sharing v_t have it replaced by v_k. Typical example



  A not–so–typically–shown example that illustrates how a mesh can fold over independent of the geometry of the mesh.



In the top collapse, the triangles are counterclockwise ordered as $\langle 0,4,3\rangle$, $\langle 4,1,2\rangle$, and $\langle 4,2,3\rangle$. The collapse of vertex 4 to vertex 0 leads to deletion of $\langle 0,4,3\rangle$ and modification of $\langle 4,1,2\rangle$ to $\langle 0,1,2\rangle$ and modification of $\langle 4,2,3\rangle$ to $\langle 0,2,3\rangle$. Both modified triangles are visible in the figure as counterclockwise.

In the bottom collapse, the modified triangle $\langle 0,2,3\rangle$ is counterclockwise (by design, collapses always preserve this), but the triangle appears to be clockwise in the figure (upside down, it folded over).

Avoid the problem by doing a look–ahead on the collapse. If any potentially modified triangle causes a folding (application specifies normal–angle threshold), assign infinite weight to the offending edge.

To avoid shrinking of mesh, assign infinite weights to boundary edges.

To preserve topology, assign infinite weights to edges with three or more shared triangles.

Data Structures. It is sufficient to store the following information about the mesh.

```
Vertex =
{
    int V;  // index into vertex array
    EdgeSet E;  // edges sharing V
    TriangleSet T;  // triangles sharing vertex
}

Edge =
{
    int V0, V1;  // store with V0 = min(V0,V1)
    TriangleSet T;  // triangles sharing edge
    int H;  // index into heap array
    float W;  // weight of edge
}

Triangle =
{
    int V0, V1, V2;  // store with V0 = min(V0,V1,V2)
    int T;  // unique triangle index
}
```

An insert operation modifies the appropriate data structures (creates new components only when necessary). A remove operation also modifies the data structures and deletes components only when their reference counts decrease to zero.

The heap is implemented as an array of pointers to `Edge` objects. It is initialized just as for polylines. An iteration is made over the edges in the mesh and the heap array values are filled in. An initial sort is made to force the array to represent a min heap.

The edge collapse is

```
void EdgeCollapse (int VKeep, int VThrow)
{
    for each triangle T sharing edge <VKeep,VThrow> do
        RemoveTriangle(T);

    for each triangle T sharing VThrow do
    {
        RemoveTriangle(T);
        replace VThrow in T by VKeep;
        InsertTriangle(T);
    }

    // Set of potentially modified edges consists of all edges
    // shared by the triangles containing the VKeep.  Modify
    // the weights and update the heap.
    EdgeSet Modified;
    for each triangle T sharing VKeep do
        insert edges of T into Modified;

    for each edge E in Modified do
    {
        compute weight E.W;
        update the heap at index E.H;
    }
}
```

During the insert and remove of triangles, edges are inserted and/or deleted in a *weak* sense. Multiple attempts are made to insert an edge shared by two modified triangles. Each time the attempt occurs, the offending triangle had changed, so the edge weight changes. To reduce code complexity, just allow the edge weight to be updated each time rather than trying to minimize the number of updates. Of course, if an edge is inserted the first time, its weight is newly added to the heap.

When an edge is deleted, it must be removed from the heap. However, the edge might not be at the root of the heap. Artificially set the weight to be $-\infty$, call the heap update to bubble the edge to the root, then remove it.

Vertices are also deleted (and sometimes inserted). Although the edge collapse makes it appear as if only the throw vertex is deleted, others can be. After each collapse, you can store the deleted vertex indices in an array that eventually represents the permutation for reordering vertices.

The function that removes triangles can be set up to store an array of the deleted triangle indices for use in reordering the triangle connectivity array.

After all edge collapses, you can build the collapse records

```
CollapseRecord
{
    int VKeep, VThrow;
    int VQuantity;  // vertices remaining after collapse
    int TQuantity;  // triangles remaining after collapse

    // connectivity indices in [0..TQ-1] that contain VThrow
    int IQuantity;
    int Index[];
}
```

- **Dynamic change in LOD.** Each edge collapse in the triangle decimation generated a set of deleted vertices and a set of deleted triangles. This information can be used to generate a sequence of records representing the collapses. The sequence can be used at run time to change the LOD. Just as for polylines, sort the triangle connectivity array (array of triples of vertex indices) so that the last triangles in the array are the first triangles deleted by an edge collapse. Sort the vertices so that the last vertices in the array are the first vertices deleted by an edge collapse (requires permuting the indices in the triangle connectivity array).
  - **Decrease LOD.** Decrement triangle–triple quantity by the amount stored in the corresponding record. Replace the appropriate indices in first part of array by the index of the deleted vertex.
  - **Increase LOD.** Increment triangle–triple quantity by the amount stored in the corresponding record. Restore the appropriate indices in first part of array. This requires remembering where you changed the indces with each collapse. Can compute this mapping once (at decimation time), then just use during run time.

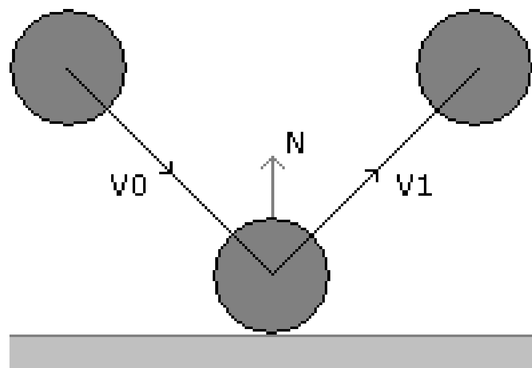
  The vertex reordering supports batch transforming of contiguous blocks of vertices and avoids having to repack data for the renderer each time the LOD changes.

Section 20. Game Physics (not in book).

- Motivation by a couple of examples.

Game Physics. Currently a somewhat nebulous term. In its simplest form, the term is used to refer to *collision response*, how a rigid object change its behavior after a collision. The response does not necessarily have to be modeled according to real physics. More complicated is to try to impose physical models involving equations of motion, including concepts such as angular velocity, angular momentum, friction, dissipation of energy, elasticity, etc.

Example. Consider a rigid sphere moving with constant velocity in a room with rigid walls. If the sphere collides with a wall, how should the direction of motion change? Similar analysis for balls on a billiards table that intersect the edges and intersect each other.



The sphere has initial center $\vec{C}_0$ and radius R and travels with constant linear velocity $\vec{V}_0$. The center path is $\vec{C}_0 + t\vec{V}_0$ for $t \geq 0$. At time $T > 0$, the sphere just touches a wall contained by a plane $\vec{N} \cdot \vec{X} = d$ where $\vec{N}$ is a unit vector. The contact time is the solution to $\vec{N} \cdot (\vec{C}_0 + T\vec{V}_0) - d = R$, or

$$T = \frac{R - (\vec{N} \cdot \vec{C}_0 - d)}{\vec{N} \cdot \vec{V}_0}.$$

This does assume that the sphere is moving towards the wall, $\vec{N} \cdot \vec{V}_0 < 0$, and that the center is initially more than R units of distance away from the wall, $\vec{N} \cdot \vec{C}_0 - d > R$. Assuming the angle of incidence is equal to the angle of reflection,

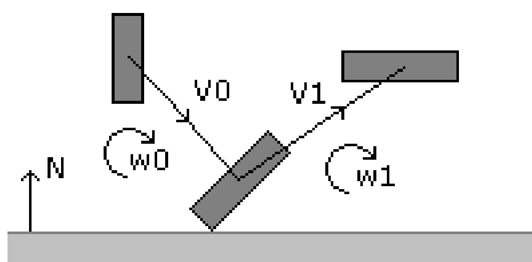
$$\vec{V}_1 = \vec{V}_0 - 2(\vec{N} \cdot \vec{V}_0)\vec{N}.$$

The path of the center is

$$\vec{C}(t) = \begin{Bmatrix} \vec{C}_0 + t\vec{V}_0, & t \in [0, T] \\ \\ \vec{C}_0 + T\vec{V}_0 + t\vec{V}_1, & t > T \end{Bmatrix}$$

If you want the energy to dissipate as a result of the collision, select a factor $\lambda \in (0, 1)$ and use $\lambda\vec{V}_1$ as the resulting velocity. Do so after each collision. Eventually the velocity is close enough to zero that you should clamp it to zero (avoids "creeping" objects).

EXAMPLE. Suppose the object is not totally symmetric, is traveling with linear velocity, but has some angular velocity that causes it to rotate about its center.



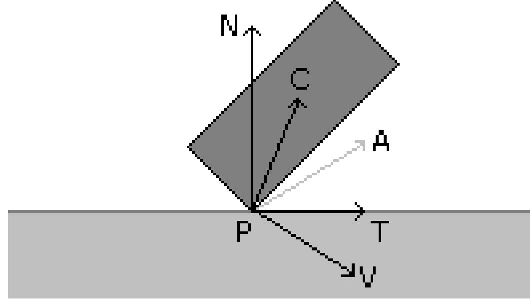
Let $\vec{C}_0$ be the center of the object. The angular velocity is represented by a rotation corresponding to a center of rotation $\vec{C}_0$ and an axis $\vec{A}_0$. The angular speed is ω_0, so the angle of rotation is $\theta(t) = \theta_0 + \omega_0 t$. The path of the center is

$$\vec{C}(t) = \vec{C}_0 + t\vec{V}_0.$$

Let $\vec{K}_0$ be the initial position of a point on the boundary of the object. The path of the point is

$$\vec{K}(t) - \vec{C}(t) = R(\vec{A}_0, \theta(t))(\vec{K}_0 - \vec{C}_0).$$

Let $\vec{P}$ be the first point of contact (what is it?) at time $t_0 > 0$.



The angular velocity should be adjusted based on the moment arm $\vec{D} = \vec{C}(t_0) - \vec{P}$ and the angular speed should be inversely proportional to size. The new axis of rotation is

$$\vec{A}_1 = \frac{\vec{N} \times \vec{D}}{|\vec{N} \times \vec{D}|}.$$

The unit–length tangent vector $\vec{T}$ that makes $\vec{A}$, $\vec{N}$, $\vec{T}$ a right–handed orthonormal coordinate frame is

$$\vec{T} = \vec{A} \times \vec{N}.$$

A choice for angular speed is

$$\omega_1 = \frac{\rho \vec{T} \cdot \vec{V}_0}{|\vec{N} \times \vec{D}|}$$

for a positive constant ρ. Observe that ω_1 is directly proportional to the speed of the object $|\vec{V}_0|$ and is inversely proportional to the size of the object (as $\vec{D}$ increases in length, ω_1 decreases in magnitude).

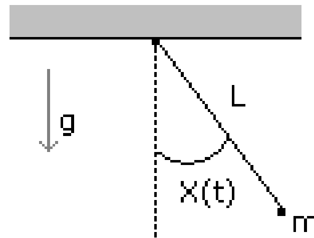
For dissipation of energy, select $\lambda_0 \in (0, 1)$ and use $\lambda_0 \vec{V}_1$ instead of $\vec{V}_1$, and select $\lambda_1 \in (0, 1)$ and use $\lambda_1 \rho$ instead of ρ.

Section 21. Game Physics (not in book).

- Physical modeling (example of pendulum).
- Numerical solution of model by explicit Euler.
- Numerical solution of model by implicit Euler.

Physical Modeling. The equations of motion are specified from physical principles. For rigid bodies, Newton's law $F = ma$ is used. For bodies whose mass changes over time, law is really $F = d(mv)/dt$ (rate of change of momentum). In a game application, the idea is to set up the differential equations that model motion and solve them. The numerical stability of a differential equation solver depends on the physical stability of the model, so the solver should be carefully chosen.

Example. A simple pendulum consists of a particle of mass m attached to a rigid wire of length L. The other end of the wire is attached to a pivot point. The pendulum is constrained to move only in a plane.



The equation of motion is

$$X''(t) + K\sin(X(t)) = 0, \quad X(0) = X_0, \quad X'(0) = V_0$$

where $K = g/(mL)$. There is no closed–form solution to this equation. For $X_0 = 0$ and small V_0, the equation is approximated by $X''(t) + KX(t) = 0$ which has solution

$$X(t) = \frac{V_0}{\sqrt{K}} \sin\left(\sqrt{K}t\right),$$

so there should be oscillatory behavior.

Euler's method provides an approximate solution to the first–order equation $X' = F(X, t)$, $X(0) = X_0$. Approximate the derivative by a forward difference

$$\frac{dX}{dt} \doteq \frac{X(t+h) - X(t)}{h}.$$

Replace in the differential equation and solve for

$$X(t+h) = X(t) + hF(X(t), t).$$

For a second–order equation $X'' = F(X, X', t)$, set $Y(t) = X'(t)$ so that $Y'(t) = X''(t) = F(X, Y, t)$. We now have a first–order system

$$\begin{bmatrix} X' \\ \\ Y' \end{bmatrix} = \begin{bmatrix} Y \\ \\ F(X, Y, t) \end{bmatrix}.$$

Euler's method provides an approximate solution

$$\begin{bmatrix} X(t+h) \\ \\ Y(t+h) \end{bmatrix} = \begin{bmatrix} X(t) + hY(t) \\ \\ Y(t) + hF(X(t), Y(t), t) \end{bmatrix}.$$

Euler's method for the pendulum problem is *unstable*. The errors accumulate so that the numerical solution leads to larger and larger oscillations about the *equilibrium* position $X = 0$.

Euler's method is an *explicit method*. The value $X(t+h)$ is given explicitly in terms of quantities at time t. These methods tend to be *conditionally stable*. *Implicit methods* tend to have better stability properties.

Approximate the derivative by a backward difference instead,

$$\frac{dX}{dt} \doteq \frac{X(t) - X(t-h)}{h},$$

and replace in the differential equation, $X(t) = X(t-h) + hF(X(t), t)$. Replace t by $t + h$ to get

$$X(t+h) = X(t) + hF(X(t+h), t+h).$$

Observe that $X(t+h)$ occurs on both sides of the equation, so it is an *implicit* term. Generally it is not possible to solve for $X(t+h)$ explicitly. Define $X_0 = X(0)$ and $X_1 = X(h)$; then $X_1 = X_0 + hF(X_1, h)$. Apply Newton's method to solve $G(X_1) = X_0 + hF(X_1, h) - X_1 = 0$ for X_1. Trade off: time for stability.

Simple code for Euler's explicit.

```
float* ExplicitEuler (float fX0, float fY0, float fH, int iN)
{
```

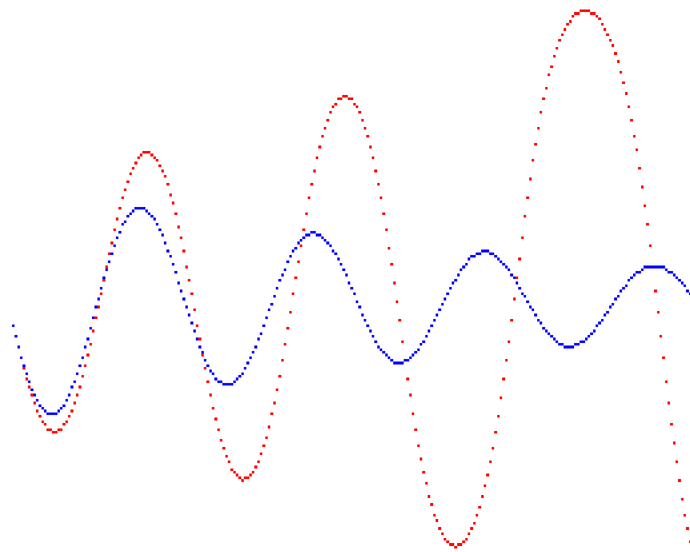
```
    float* afExplicit = new float[iN];
    for (int i = 0; i < iN; i++)
    {
        float fX1 = fX0 + fH*fY0;
        float fY1 = fY0 - fH*gs_fK*sinf(fX0);

        afExplicit[i] = fX1;
        fX0 = fX1;
        fY0 = fY1;
    }
    return afExplicit;
}
```

Simple code for Euler's implicit.

```
float* ImplicitEuler (float fX0, float fY0, float fH, int iN)
{
    const float fK0 = gs_fK*fH*fH;

    float* afImplicit = new float[iN];
    for (int i = 0; i < iN; i++)
    {
        float fK1 = fX0 + fH*fY0;
        float fX1 = fX0;
        for (int j = 0; j < 32; j++)
        {
            float fG = fX1 + fK0*sinf(fX1) - fK1;
            float fGDer = 1.0f + fK0*cosf(fX1);
            fX1 -= fG/fGDer;
        }
        float fY1 = fY0 - fH*gs_fK*sinf(fX1);

        afImplicit[i] = fX1;
        fX0 = fX1;
        fY0 = fY1;
    }
    return afImplicit;
```

```
}
```

Output for $X_0 = 0.1$, $Y_0 = 1$, $h = 0.1$, and 256 iterations.



Section 22. Game Physics (not in book).

- Linearized stability analysis
- Modal equation
- Numerical stability of solver is directly related to physical stability at equilibrium points (step size for solver cannot be chosen arbitrarily)

Linearized Stability Analysis. The (nonlinear) equation of motion can be approximated by a linear equation near each equilibrium state. Consider the simple pendulum with damping

$$X''(t) + PX'(t) + K\sin(X(t)) = 0, \quad X(0) = X_0, \quad X'(0) = V_0$$

where $P > 0$ and $K > 0$. The two equilibrium states are $X(t) = 0$ and $X(t) = \pi$. At $X(t) = 0$, the linearized equation is

$$X''(t) + PX'(t) + KX = 0.$$

The corresponding *characteristic equation* is $\lambda^2 + P\lambda + K = 0$. At $X(t) = \pi$, the linearized equation is

$$X''(t) + PX'(t) - KX = \pi.$$

The corresponding characteristic equation to the homogeneous equation is $\lambda^2 + P\lambda - K = 0$. An equilibrium solution is stable whenever the roots to its characteristic equation both have negative real parts. If both real parts are zero, you get marginal stability (undamped pendulum, for example). Assuming $P > 0$, $X = 0$ is stable since the roots are

$$\lambda = \frac{-P \pm \sqrt{P^2 - 4K}}{2}.$$

Both roots are negative real numbers when $P^2 \geq K$ or have negative real parts when $P^2 < 4K$. Equilibrium solution $X = \pi$ is unstable since the roots are

$$\lambda = \frac{-P \pm \sqrt{P^2 + 4K}}{2}.$$

Both roots are real–valued, but one is positive and one is negative.

Modal Equation. Let $\vec{U}(t) = (X(t), X'(t))$. For each root λ, the corresponding *modal equation* is

$$\vec{U}'(t) = \lambda\vec{U}.$$

Define $h > 0$ to be the step size of a numerical solver. Define $\vec{U}_n = \vec{U}(t+nh)$ and $E\vec{U}_n = \vec{U}_{n+1}$. Numerical methods for solving the modal equation are typically of the form

$$P(E)\vec{U}_n = 0$$

where $P(E)$ is a formal polynomial in the operator E. For a complex variable z, you can analyze the roots of $P(z) = 0$. The numerical method is stable whenever $|z_k| \leq 1$ for all roots z_k of P.

The explicit Euler's method for the modal equation is

$$\vec{U}_{n+1} = \vec{U}_n + h\lambda\vec{U}_n$$

or

$$[E - (1 + h\lambda)]\vec{U}_n = 0.$$

The polynomial is $P(z) = z - (1 + h\lambda)$ and has root $z = 1 + h\lambda$. For stability we need $|1 + h\lambda| \leq 1$.

The implicit Euler's method for the modal equation is

$$\vec{U}_{n+1} = \vec{U}_n + h\lambda\vec{U}_{n+1}$$

or

$$[(1 - h\lambda)E - 1]\vec{U}_n = 0.$$

The polynomial is $P(z) = (1 - h\lambda)z - 1$ and has root $z = 1/(1 - h\lambda)$. For stability we need $|1 - h\lambda| \geq 1$.