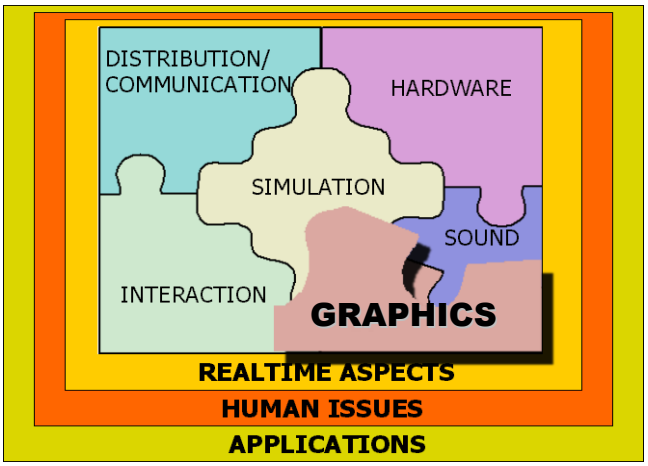
VR - Real Time Graphics



The ”real” Virtual Reality

- What does it take to create a virtual world indistinguishable from the real, in the visual sense?
  - Using polygons, the world can be represented with 80 million polygons.

    Alvy Ray Smith, 1974 (Microsoft Graphics Guru)
- With 30Hz we get 2.4Billion polygons/sec.
- With 8 SGI IR3 we can get 214 Million/sec.
- A screen with 1600x1400 pixels we have 2.2 million pixels.
- So even if we can generate the pixels, we can’t display them (yet)

Real time ”A Bugs Life”

- 1998 Pixar/Walt Disney released A Bugs Life.
- Generating an average frame took 3-4 h (even up to 20h) on a 330Mhz UltraSPARC
- A minimum frame rate of 12 fps would take a speed-up of 150.000 times.
- Given Moore’s law, (CPU acceleration rate of 2 times every 18 months)
  - would get us to year 2024.
- Graphic chips have developed faster than that.
- Graphic chipsets are doubling in every 6 months.
  - Would lead to 2007
- Very unlikely that it can continue like that.

Real Time ”A Bugs Life”

- Electron leakage and Quantum effects will appear when the electronics is shrinked far enough.
  (Colin P. Williams et. al., *Explorations in Quantum Computing)*
- Methods for optimizing the graphics, both in software and later in hardware will make performance take leaps (maybe).
- So lets look at some methods for getting most of the graphic card.

Real Time Graphics

- Frame-rate - so very important for VR
  - Low frame-rate:
    - latency for the user
    - No presence
  - Uneven frame-rate
    - Due to different object detail in the scene
    - Jerky graphics
    - Hard to interact with objects when speed in unpredictable
  - Constant high frame-rate - To wish for!!
- 6 fps, a sense of interactivity
- 15 fps is real-time
- 30 fps near optimal
- > 72 fps changes in display rate is in-detectable

Real Time Graphics

- Why special purpose hardware for graphics?
  - Example: 1E6 vertices gives us 333333 polygons (if no sharing is allowed). Each vertice consists of a (x,y,z) tuple. A transformation consists of a 4x4 matrix. Transforming this geometry will lead to 1E6 matrix multiplication's + lightning, fog, transparency, textures, …
  - Even transmitting the geometry over the bus takes time
    - Triangle requires 36 bytes (normals, texture coords, coords)
    - 1 Million polygons
    - 36 Mb/s only for geometry
    - 30 fps -> 1080 Mb/s throughput between processor and graphics hardware.
  - We need special hardware for an efficient handling of this.

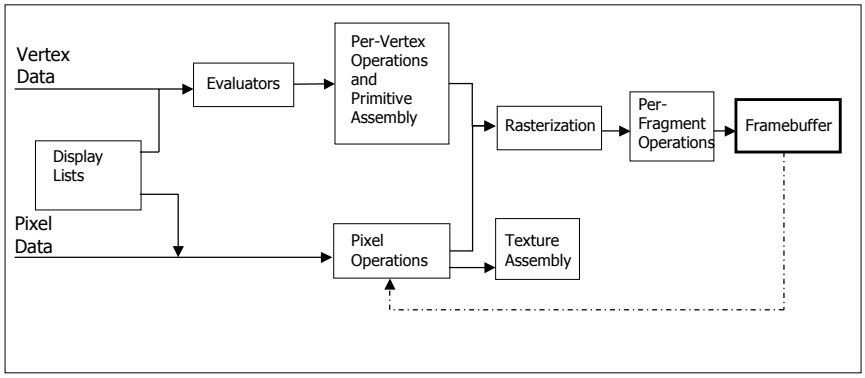
Graphics Architecture

- Application Stage
  - Input
    - User interaction, Simulations
  - Output
    - Object transform, View transform, Geometries
- Geometry Stage
  - Lighting, clipping, screen mapping
  - Output
    - Lighted, textured polygons in screen space
- Rasterizer
  - Scan-line conversion
  - Output
    - Pixels in frame buffer

The Geometry Stage

- Processes vertices and pixels, generating screen mapped polygons.

Example: The OpenGL pipeline



The OpenGL pipeline

- Display lists
  - Geometry's and their operations can be done static, which leads to that the HW can optimize the rendering for these objects.
- Evaluators
  - Polynomial mapping. Surfaces are evaluated from control-points. Eg. NURBS.
- Per-vertex operations
  - Conversion of vertices to primitives (eg. Triangles). Texture normals and coordinate generation. If lightning is used, there will be light-calculations.

The OpenGL pipeline

- Primitive Assembly
  - Clipping, perspective calculations, culling.
- Pixel Operations
  - Scaling, mapping, clamping
- Texture assembly
  - Special hardware for texturing for extra speed.
- Rasterization
  - Converts geometries and pixel-data to fragments. Each pixel corresponds to a pixel. Antialiasing.

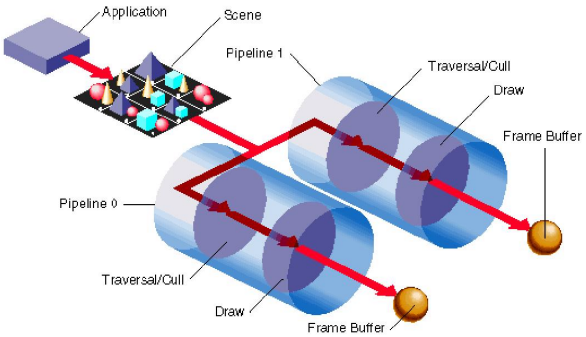
The OpenGL pipeline

- Fragment operations
  - Fog, texture, alpha test, scissor test, depth-buffer test, blending, bit-masking
- Framebuffer
  - Contains the image that will be displayed on the screen.
  - Frontbuffer - The visible image.
  - Backbuffer - The image that is rendered in background.
  - Swap buffer - Switches the front and the backbuffer.
  - Left/Right - For left and right eye.

Infinite Reality 2

- Onyx2 IR2 - optimized for real-time graphics
- Highly parallel.

Infinite Reality



Infinite Reality

- 10 Million textured polygons/second
- Up to 128 MIPS R10000 Processors
- Up to 16 InfiniteReality2 graphic pipes
- Up to 256 GB primary memory
- Up to 4 Raster Managers (RM)/pipe
- Up to 64 MB texture memory/RM
- 711MB/s CPU-CPU bandwidth
- 320MB frame buffer

- "RealityMonster™ multisubsystem rendering mode provides up to 210 million polygons per second and 7.2 gigapixels per second fill rate or 1GB of physical texture memory (with 16 pipes) for tackling grand-challenge applications"

Infinite Reality 3-4

- MIPSR12000
- 512 Mb Texture Cache
- 40-50 Million textured polygons/second

Others?

- XBox from Microsoft
  - Processor from Intel
  - Graphics from NVIDIA
    - 6.4GB/sec Memory Bandwidth
    - 150 million micropolygons/particles per second
    - 150 million transformed and lit polygons per second
    - 100 million polygons per second sustained performance (shading, texturing)
    - 4 simultaneous textures
    - Compressed textures available at 6:1 compression
    - Full-scene anti-aliasing
    - DVD movie playback
    - 1920x1080 maximum resolution
    - HDTV support
- Playstation™ II
  - 75 Million polygons/second (with lighting, textures, z buffering, and alpha blending)

Cheating?

- To get the visual result we want to use more polygons and larger textures.
- Remember, **real-time!**
- The world has a constant level of detail, no matter how far away we are from a object, it always has the same resolution.
- Think of a tree, all the leaves. A lot to render.
- This leads to some tricks for getting the speed up!
- Lets first take a look of how the graphical scene is built up

Procedural vs. Scene graph

- Procedural
  - Pure OpenGL
  - Sequence of commands
  - Commands repeatedly executed
- Scene graph
  - Setup data structures
  - Render by parsing data structures

Scene-graph

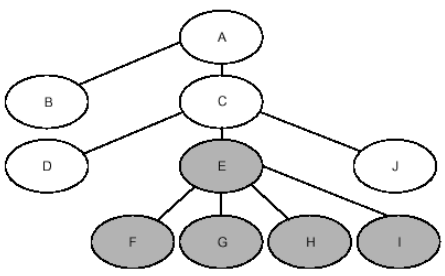
- Defines the scene that will be rendered.
- A hierarchical arrangement of graphical entities to create the visual scene
- Directed acyclic Graph (DAG)
- Possible for the graphic engine to optimize
  - View frustum culling usually implemented
    - You have to group objects that are spatially close in the same sub-branch of the tree!
- Usually traversed in depth-first order
- Common for most graphic API:s
- Suits a bottom-up design of objects
- Behavior can be built into the scene-graph, Collision detection, animations, …
- Remarks about OpenGL:
  - OpenGL is not a scene-graph, scene-graphs is usually built upon it.
  - OpenGL is a graphic standard for low-level graphics.

Scene-graph

- Raises the programming ”floor”
- Thinking objects
  … not vertices
- Thinking content
  … not rendering process
- It is descriptive
- A scene is a collection of geometries, lights, sound, … Together with positional information that place these elements at particular positions.
- Provides the hierarchical framework for grouping objects together, spatially.
- Some definitions:

Scene-graph

- **Root node** Each scene graph has only one *root node*. The root node in figure is node A.
- **Scene graph tree** All of the nodes in a scene graph, arranged in a hierarchical order. The nodes in figure are all in one *scene graph tree*.
- **Sub-tree** A node and all of its descendants. A *sub-tree* in figure is shaded.
- **Sibling** Children of the same parent node are *siblings*. Nodes F, G, H, and I are siblings.
- **Traversal order** The order in which nodes in a scene graph are processed while the simulation is running. The nodes in the scene graph in figure have been labeled so that their alphabetical order indicates the proper traversal order.



Scene-graph

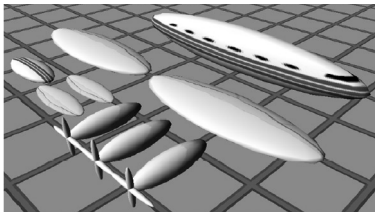
- **Ancestor** Since node A has a sub-tree that includes node E, it is an *ancestor* of node E. Note that node J is not an ancestor of node I.
- **Child node** A node's direct descendant. In figure 4-3, nodes B and C are both children of node A. J is not a *child* of A.
- **Descendant** Any node that is contained in the sub-tree of another node is considered to be a *descendant* of that node. In figure, node F is one of the descendants of node A.
- **Leaf node** A node without children. Nodes B, D, F, G, H, I, and J are all leaf nodes.
- **Parent node** A node's direct ancestor. Node A is a *parent* of node C, but not of node E. It is possible for a node to have several parents.
- **Predecessor** Since nodes B and C are processed before node J, they are its *predecessor*s. A node's predecessors can affect the rendering of that node, even though they may not be ancestors.

Scene-graph

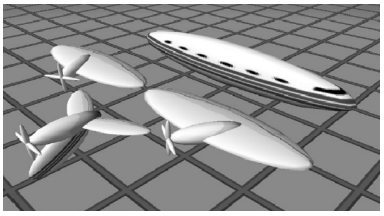
- Example of some scene-graph elements:
  - Group Nodes
    - Collects a number of nodes in a logical way
    - Level of Detail, Switch nodes
  - Transformation Nodes
    - Moves, rotates, scales its children/siblings
  - Light Nodes
    - Make geometry's lit
    - Pointlight, Spotlight, Directional, ...
  - Others
    - Sound, fog, manipulators (animators), Collision, ...
  - Geometry nodes
    - Polygons, lines, points, ...
    - Contains the actual polygons that will be rendered

An example

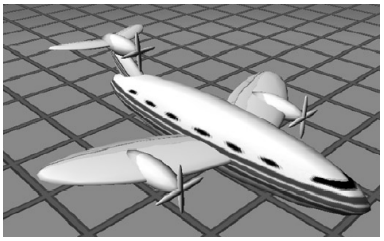
Scene graph items



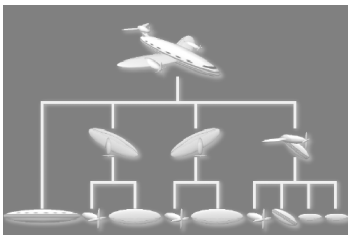
Scene graph groups



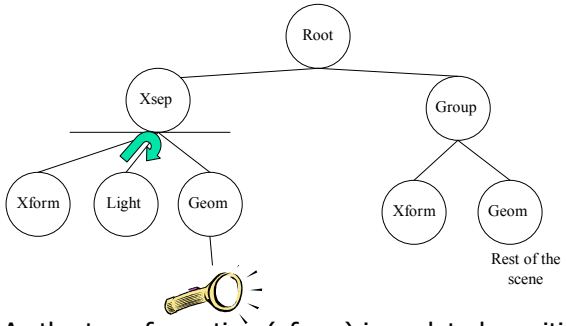
Final Scene



The Scene graph



Scene graph (WTK- example)



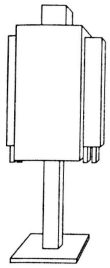
- As the transformation (xform) is updated, position/orientation of the light will also change.

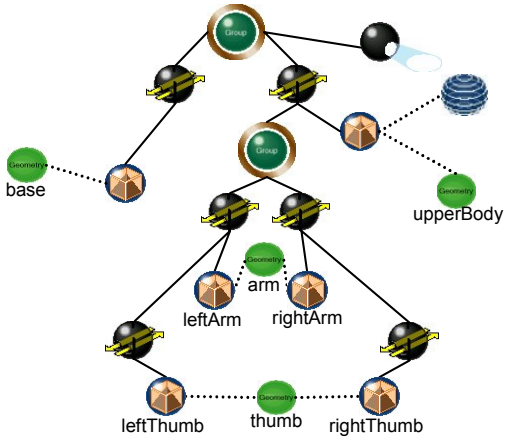
Scene graph

- A graphic API usually contains the math associated with graphics
  - 2D/3D vectors
  - 3x3/4x4 Matrices
  - Vector/matrix math (inverse, add, …)
  - Intersection
    - Ray-ray
    - Ray-polygon
    - Point on plane
    - Box-box
- Example of some API:s with scene-graphs
  - Java3D
  - Cosmo3D
  - WTK
  - Performer
  - Inventor

Scene graph

- Example in Cosmo3D





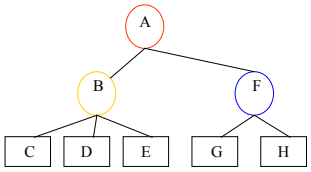
Scene graph

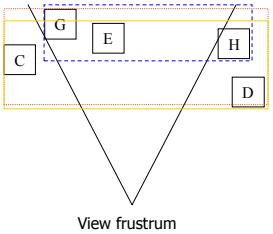
- Remarks
  - Designers of an API can't foresee all necessary functionality
  - Therefore a Scene Graph API should be:
    - Extendable
      - Possible to modify all functionality, replace with own code.
    - Add new types of traversal tasks
      - Collision detection
      - Occlusion culling
      - Add new scene abstractions
      - Advanced shading
      - Surfaces
      - Progressive meshes
  - **Cheasy Extensibility**
    - Add "user-data" pointers to nodes
      - Store additional application-specific data at node
    - Add traversal callbacks to nodes
      - Perform application-specific processing at each node visit
    - Hard to share with other developers
    - Extensions embedded in the application

Scene graph

- Always draw the scene-graph on paper for optimization
- Hard to see the structure in Code!

Inefficient spatial arrangement of nodes





View frustrum

1. Is A:s bounding area visible in any part? - YES, check all its children (B and F)

    1.1. Is B:s bounding area visible in any part? - YES, check all its children (C, D and E)

        1.1.1 Is C visible - NO

        1.1.2 Is D visible - NO

        1.1.3 Is E visible - Yes, RENDER IT

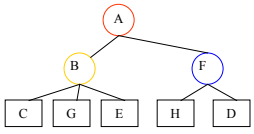
    1.2 Is F:s bounding area visible in any part? - YES, check all its children (G, H)

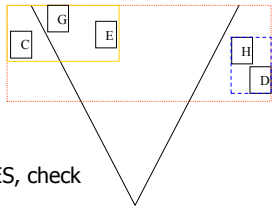
        1.2.1 Is G visible? - Yes, RENDER IT

        1.2.2 Is H visible? - No

→ 8 calculations for this scene.

Efficient spatial arrangement of nodes





1. Is A:s bounding area visible in any part? - YES, check all its children (B and F)

    1.1. Is B:s bounding area visible in any part? - YES, check all its children (C, G and E)

        1.1.1 Is C visible - NO

        1.1.2 Is G visible - Yes, RENDER IT

        1.1.3 Is E visible - Yes, RENDER IT

    1.2 Is F:s bounding area visible in any part? - NO

→ 6 calculations for the same scene.

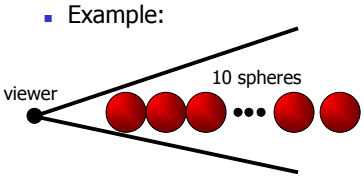
So by arranging the spatially close nodes close also in the scene-graph hierarchy we can get more efficient rendering.

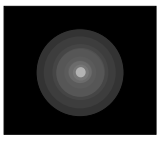
Culling

- To Cull – "Select from a flock"
- Backface culling
  - Don't bother rendering polygons facing off the viewer.
  - Culling reduces the load on the rasterizer, but increases the load on the geometry stage because the backface computation are done there.
- Hierarchical View frustum Culling
  - Using *Bounding Volumes* (BV) to enclose objects
  - BV organized in a bounding volume hierarchy
  - View frustum is recursively compared against BV hierarchy.
  - Objects outside the view frustum will not be rendered
  - Scene graphs is used to store the BV:s
  - HVFC occurs in application stage, geometry and rasterizer stage can benefit enormously

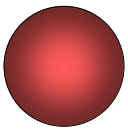
Culling

- Occlusion culling
  - Visibility can be solved either with hardware (Z-Buffer)
  - This is not really smart as:
    - All objects has to be scan-converted and compared to Z-buffer
    - After comparison they can be written to color-buffer and Z-buffer.
    - Example:





Deth complexity



Rendered image

  - 9 spheres are rendered unnecessarily
  - Occlusion algorithms try to avoid this.

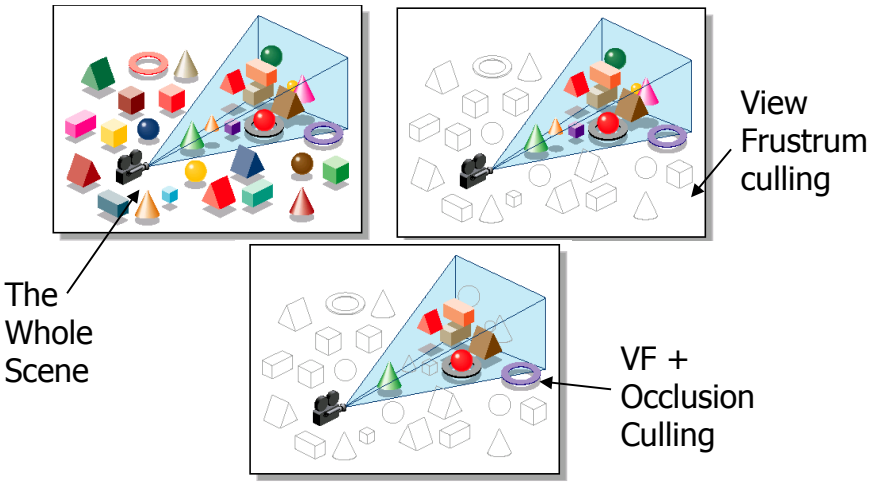
Occlusion Culling

- Cull away objects in the view frustum but not visible in the final image.

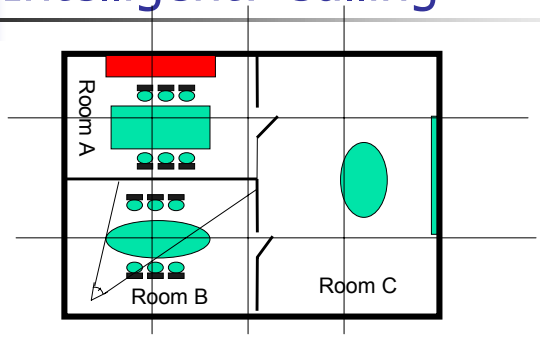
G contains all objects in the scene. O_R is the occlusion representation

```
1:  OcclusionCullingAlgorithm(G)
2:  O_R=empty
3:  for each object g ∈ G
4:  if (isOccluded(g, O_R))
5:          Skip(g)
6:      else
7:          Render(g)
8:          Update(O_R, g)
9:      end
10:end
```


Culling



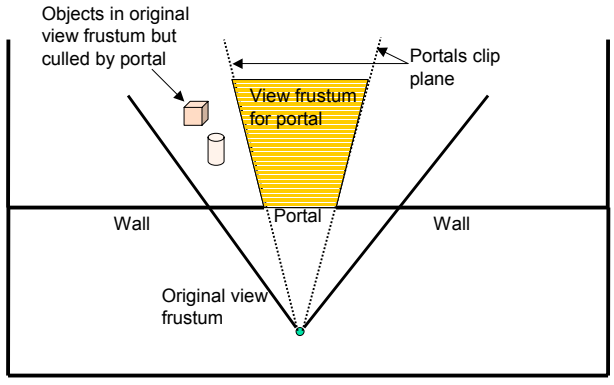
Intelligent? Culling



- Involves some logic thinking when constructing the application.
  - No need to render Room B&C when viewer are in room A with door closed.
  - Divide the scene in boxes (volumes)
    Calculate which boxes (volumes) that are visible from my point of view.

Portal Culling

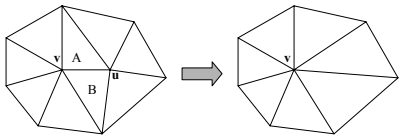
- This of course don't work in an outside environment.
  - Many game-engines problem.



Numerical Design Ltd, NetImmersion game engine

Detail Culling

- Sacrifice quality for speed
- Optimizing geometries. I.e.. Removing unnecessary polygons.
- Mesh Simplification
  - The process of taking a complex model and reducing it´s complexity.
  - **Batch oriented**, i.e.. The geometry is prepared before it is rendered in the application.
  - Creating a set of discrete LODs.
    - This is done by *edge collapsing*.



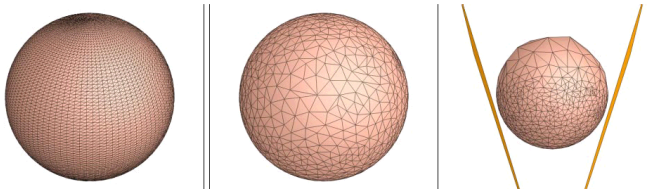
- Left shows figure before ***uv*** edge collapse occurs; right shows point **u** collapsed into point **v,** thereby removing triangles A and B and edge **uv**.
  - Michael Garland, Paul S. Heckbert, Carnegie Mellon University
  - Hugues Hoppe, Microsoft Research.

Detail Culling

- *Geometric algorithms*
  - *Force reduction of the data*
  - *Cluster multiple vertices of the polygonal object that are close in space into one.*
  - *Remove all triangles that degenerate or collapse in the process.*
  - *No control over local detail*
  - *Robust on all kinds of in-data.*
  - *E.g.. Hierarchical Structures for Dynamic Polygonal Simplification.* *(Jarek Rossignac, North Carolina Chapel Hill)*

Detail Culling

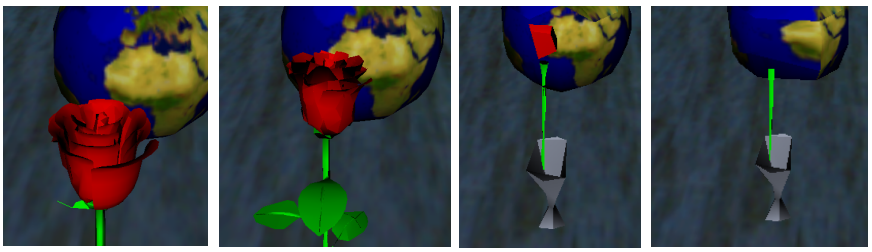
- **Geomorph LOD:s - "On the fly"**, the geometry is calculated during rendering
- The process of mesh simplification have an interesting property that allows of making a transition between LODs.
  - Each generated model has fewer
  - *View-Dependent Refinement of Progressive Meshes,* (Hughes Hoppe, Microsoft Research)
    - Show only what we can see in full resolution, the rest can be displayed very coarse.



Original (19800 faces) Front view (1422 faces) Top view (1422 faces)

Levels of Detail (LOD)

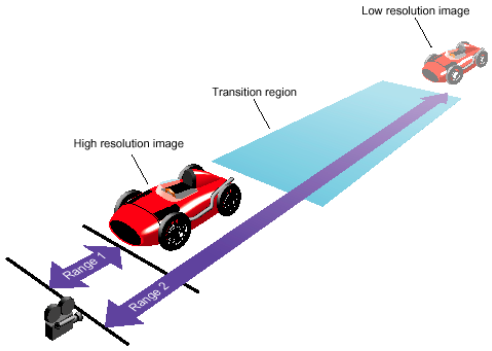
- Multi level geometry's
  - Usually programmer sets up the distances where the transition between two LOD:s occur.
  - Discrete
    - Draw a coarse model when the viewer are far away from the object. Switch for a more accurate when we get closer.



Levels of details generated with: Lodestar - A VRML 1.0 LOD generator

Levels of Detail

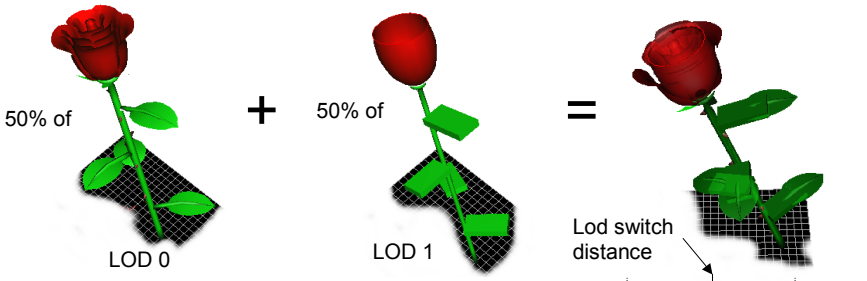
- Problem with ”popping” between discreet models



Levels of Detail

- Alpha LOD
  - The same as above but with the switching made using graduate transparency.
  - + Smooth transition between two lod:s.
  - - Both geometry's has to be rendered during the transition - performance penalty.
  - - Transparency can introduce artifacts (unless polygons is sorted in back-front order).

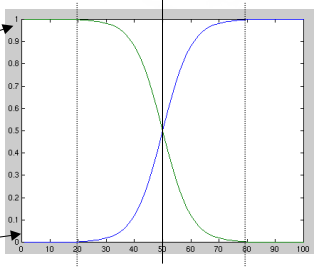
Alpha LOD



$$T_{lod_n} = \frac{1}{1+e^{-\beta(x-\alpha)}}$$

$$T_{lod_{n+1}} = 1 - \frac{1}{1+e^{-\beta(x-\alpha)}}$$

} Transparency functions



Levels-of-detail

- Geomorph LODs
  - Parameterize the model so the number of polygons are calculated from the viewing distance.
  - Smooth Levels Of Detail (Vienna University if Technology, Austria)
  - *View-Dependent Refinement of Progressive Meshes,* (Hughes Hoppe, Microsoft Research)
- Problems with texture can occur for automatic levels-of-detail generators.

Constant frame rate

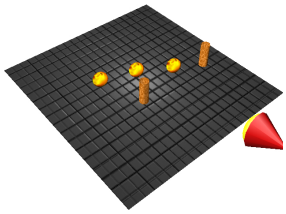
- As told before, very important.
  - Step 1: Optimize the scene
  - Step 2: Use all tricks possible when rendering objects
  - Step3: Use dynamic clip-plane
  - Step 4: Dynamically change the lod on the geometry's
- Dynamically moving the back-clipping-plane
  - Move the back-clipping plane towards viewer when frame-rate drops.
    - Could lead to that most objects are not rendered at all when viewing detailed parts of the scene.
    - Could remove immersion.

LOD Management

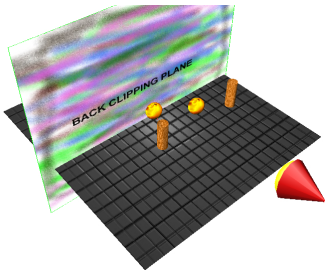
- Dynamically change the LOD on the geometry's
  - When frame-rate drops, change the level of detail on the geometry's to a more coarse version (less detail).
  - Start with objects in the viewers periphery?
  - The same for objects far away
  - Selecting the "best" models is a "knapsack problem", how to decide which models to operate on?
  - Funkhouser Séquin[1] presented a heuristic algorithm adapts LOD for all visible objects to obtain constant frame-rate.

1. Adaptive Display Algorithm for Interactive frame rates During Visualization of Complex Virtual Environments, Computer Graphics (Siggraph '93 Proceedings)

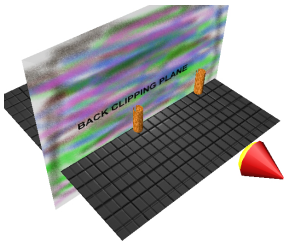
Constant frame-rate



The whole scene
Target frame-rate: 30fps
Actual frame-rate: 15fps



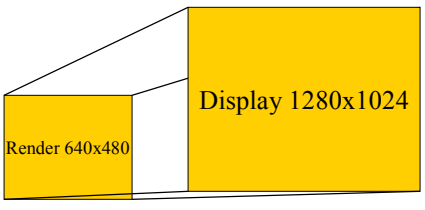
Clip-plane moved towards viewer
Target frame-rate: 30fps
Actual frame-rate: 25fps



Clip-plane moved even more towards viewer
Target frame-rate: 30fps
Actual frame-rate: 31fps

Constant frame-rate

- Performer - A high performance graphics API for Irix (Linux, Moongoose).
  - Uses dynamic video-resolution (DVR)
  - Works on Infinite Reality and IR2
  - Renders the scene in lower resolution which is zoomed up to full output resolution using bilinear interpolation without added latency.



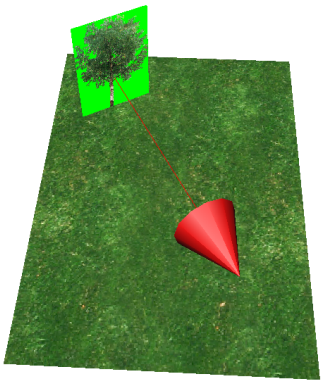
Billboard

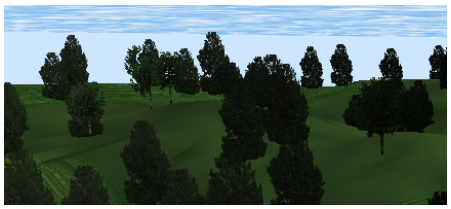
- Instead of rendering all the leaves in a tree, just use a photograph of a tree, texture a flat polygon and make the polygon always facing the viewer.
- Often used as avatars!
- Example:

```
orientation = viewpoint->getOrientation()
billboard->setOrientation(orientation)
```

Or even better:

```
vec = viewer->getPosition() - billboard->getPosition()
billboard->setOrientationVec(vec)
```





Radiosity

- An algorithm to realistically render a scene off-line.
  - Done before running the application
  - Calculate shadows, lighting, textures, …
  - Very nice result
  - Outputs colored polygons or large textures.

References

- *Real-Time Rendering*, Tomas Möller, Eric Haines, A K Peters, 1999
- *Migrating to an Object Orientated Graphics API*, Course notes from Siggraph 2000.
- *Developing Efficient Graphics Software: The Yin and Yang of Graphics*, course notes from Siggraph 2000.
- *Approaches to Polygonal Model Simplification*, Course notes from Siggraph 2000
- *Advanced Animation and Rendering Techniques*, Alan Watt, Mark Watt, Addison Wesley, 1992