Crystal Space

An Open Source and Portable 3D Engine
http://www.crystalspace3d.org/

Februari 2008
By Jorrit Tyberghein

Contents

- Introduction
- Contributors
- Portability
- Modularity and Extensibility
- Data Structures and Performance Techniques
- Debugging
- The 3D Engine
- The 3D Renderer (rasterizer)
- Various other plugins
- Crystal Entity Layer
- Apricot

Intro: What is a 3D Engine?

- Core responsibilities of a 3D Engine:
  - Manage 3D objects and organize them hierarchically in a 'world'
  - Manage surface attributes of objects (materials, shaders)
  - Visibility Determination: only draw what you can see
  - Level of Detail: reduce detail when it is not needed
  - Lighting system
- Non-core responsibilities:
  - Collision detection and physics
  - Scripting
  - AI
  - Sound
  - ...

Intro: Problems

- While developing a 3D engine various problems need to be solved:
  - Supporting the latest hardware features and still working well on low-end hardware
  - Finding efficient data structures
  - Finding efficient algorithms
  - Portability: different hardware, different OS, different window system, ...
  - Classical programming problem: debugging

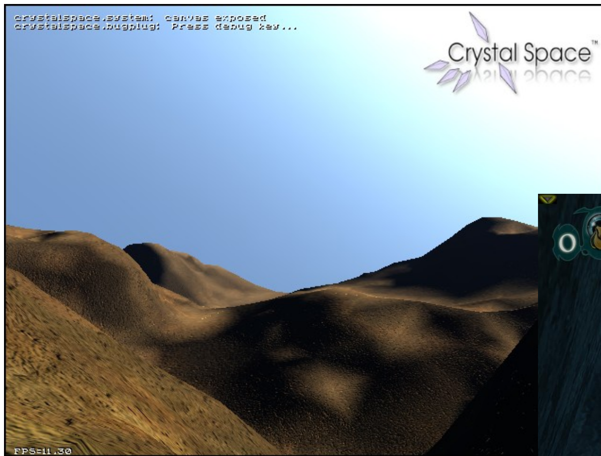
Intro: What is Crystal Space?

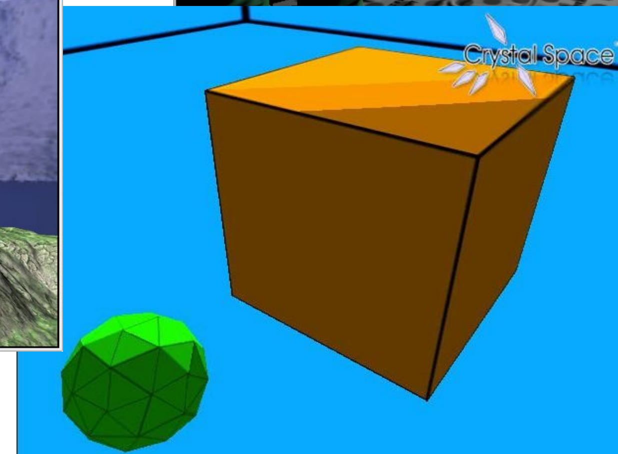
- Crystal Space is a 3D application framework
  - Not specific to games
  - Open Source
  - Portable
  - Modular
  - Extensible
  - Feature rich
  - Support for low-end and high-end hardware
- Crystal Entity Layer sits on top of Crystal Space:
  - Adds game logic

Intro: Crystal Space Features

- Portals
- KD-Tree based culler
- Particle Systems
- Volumetric Fog
- Landscape engine
- Shaders
- Lightmaps
- Stencil shadows
- Physics (ODE or Bullet)
- Sound
- Virtual File System
- LOD
- Procedural Textures
- Window System based on CEGUI
- ...

Intro: Crystal Space Features





Contributors

- Crystal Space started in August 1997
- More than 180 people have contributed to it since then (February 2008)
- Some statistics according to ohloh.net:
  - Codebase: 922000 LOC
  - Effort (est.): 256 Person Years
  - Project Cost: $14059167

Portability

- Goals
- Operating Systems
- CPU's
- Window Systems/API's
- 3D Hardware
- Sound systems
- Input Devices

Portability: Goals

- The developer using Crystal Space for a game or 3D application should not have ANY worries about portability. Crystal Space 'should' handle it all

Portability: Operating Systems

- Different operating systems have different API's to handle various things:
  - Getting the current system time
  - Getting the current directory
  - Getting the home directory
  - Finding user preferences
- File system works differently:
  - GNU/Linux: one root: /usr/local/CS/somefile.txt
  - Windows: several roots: c:\Program Files\CS\somefile.txt
  - CR/LF vs LF
  - Crystal Space makes abstraction: VFS
    - /gamedata/somefile.txt
- Crystal Space runs on: Windows, GNU/Linux, MacOS/X

Portability: CPU's

- CPU's have various differences:
  - Byte order: little endian vs big endian
  - Structure packing and alignment
  - Different sizes of 'int'
- Use ASCII format to write out data as much as possible (XML)
- Be careful if binary format is needed

Portability: Window Systems

- Big differences in window API
  - X Windows for Unix and GNU/Linux (and also Windows if you want)
  - DirectDraw on Windows
  - MacOS/X
- Depending on the renderer (OpenGL, software) also other window API needed (GLX, WGL, ...)
- Very hard to support all in the same code base
- Use multiple plugins (x2d, ddraw, glx2d, glwin32, ...)

Portability: 3D Hardware

- Many types of 3D cards available. A lot of differences in capabilities
  - Shader programming
  - Stencil buffer
  - Available texture memory
  - Presence of supported OpenGL extensions
  - Other hardware capabilities
- With no 3D card: software renderer is also possible
- Again using different plugins for different renderers

Portability: Sound

- Different sound API's
  - OpenAL
  - OSS
  - DS3D
  - EAX
  - Arts
- Different sound file formats
  - WAV
  - OGG

Portability: Input Devices

- Mouse, Keyboard, Joystick
- Portable event system so that applications know about various types of input without having to worry about how the OS handles it

Modularity and Extensibility

- Crystal Space is a big project: could be hard to manage
- Need to split in independent modules
- Modules are called 'plugins' in CS
- Only a few basic libraries, all the rest are plugins

Modularity: Basic Libraries

- csutil
  - SCF: plugin system, reference counting, and interface handling
  - Portable API's for system specific tasks (current time, current dir, ...)
  - Utility classes: growing arrays, string classes, hashmap, ...
- csgeom
  - Mathematical classes: matrices, vectors, polygons, ...
- csgfx
  - Some image manipulation tools, assistance for the image loading plugins
- cstool
  - High level library containing helper classes for applications to do common stuff (app initialization for example)

Modularity: Plugins

- A plugin represents some functionality. We call this a *plugin type*. Some example types:
  - Mesh object plugins: geometry for the engine
  - Sound drivers
  - Canvas plugins: interface to the window API
  - Shader plugins
  - Image loader plugins (PNG, JPG, GIF, ...)
  - ...
- Big advantages:
  - Other people can write plugins and extend CS (extensibility)
  - Every plugin is more or less stand-alone (modularity)

Modularity: Interfaces

- A plugin implements one or more *interfaces*
  - An interface is a contract
  - A plugin type is represented by one or more interfaces. A plugin of a plugin type must implement at least those interfaces
  - An interface is similar to interfaces in Java. SCF implements this for CS in C++
  - The set of all interfaces form the Crystal Space API

Modularity: Interface Example

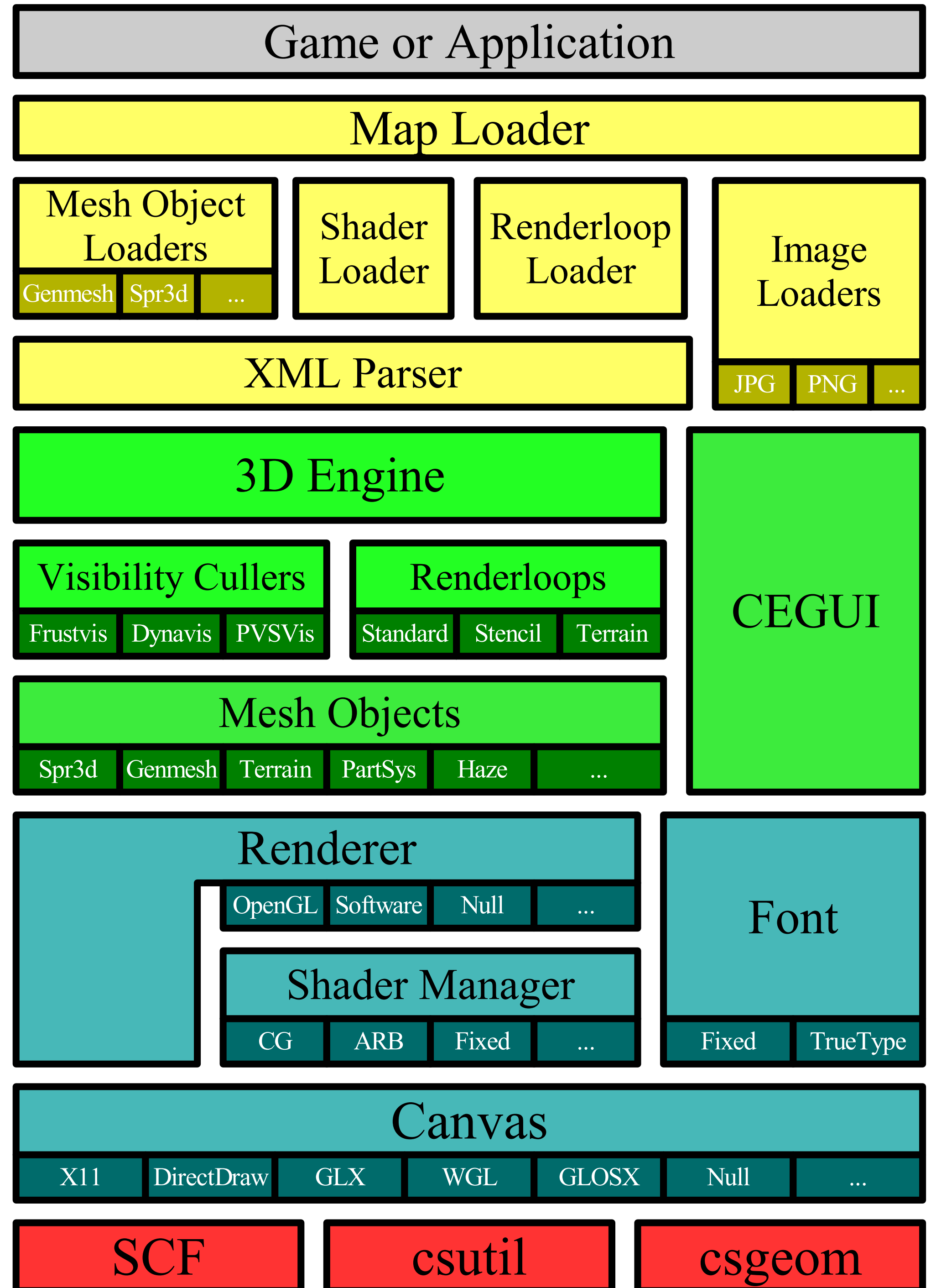
- Example interface in SCF:

```
struct iLight : public virtual iBase
{
    SCF_INTERFACE(iLight,0,0,1);

    /**
     * Set the color of the light. Unless the light is dynamic
     * this doesn't have an immediate effect.
     */
    virtual void SetColor (const csColor& color) = 0;

    /**
     * Get the current color.
     */
    virtual const csColor& GetColor () const = 0;
};
```

Modularity: Architecture Overview



Data Structures and Algorithms

- Fast computers and modern 3D cards are important for performance

  BUT

- Without efficient data structures and algorithms all that fast hardware is wasted

Data Structures: Memory Management

- Allocating and deallocating memory is expensive
- Ways to deal with that:
  - Avoid memory management in critical code (tight loops)
  - Try to reuse memory that you allocated previously: use pools
    - Instead of freeing memory you put pointer back on a pool
    - At allocation time you only allocate if you have no more free items in the pool
  - Try to avoid memory allocation/deallocation: use block allocator
    - Allocate multiple instances of an object in blocks (i.e. 1000 at same time)
    - Have a simple linked list of free objects in every allocated block to quickly find free space

Data Structures: Growing Arrays

- Linked lists are almost always bad:
  - Every node in a linked list has memory overhead (for next and possible prev pointer)
  - Additional memory overhead imposted by operating system for every memory allocation (can be very big)
  - Memory allocation/deallocation is slow
  - Every node must be allocated separately
  - No direct indexed access possible
- Growing arrays are often better:
  - Every element in an array is just as big as it needs to be (ignoring packing overhead)
  - Allocation happens in groups
  - Direct access

Data Structures: Lazy Processing

- It's good to be lazy!
- General principle: avoid doing something until you really need to do it
  - Sometimes it turns out you don't even need to do it at all
  - Several actions can be combined in one processing run
- Example:
  - When moving an object in the kd-tree don't rebuild the affected part of the tree until you need to go there: if you have several movements of the object before the tree is needed then you only have to rebuild the sub-tree once. Also movements of different objects in the same sub-tree will not cause multiple sub-tree rebuilds

Debugging

- Debugging takes 90% of the time (no exaggeration!)
  - Common errors:
    - Forgetting to initialize variables/memory
    - Overwriting memory you don't own (writing past array boundary)
    - Double free of memory
    - Memory leaks (forgetting to free memory)
    - Mathematical errors (division by zero and others)
    - Logic (algorithm) errors
  - For the first four errors there are tools to help debug this (valgrind on linux, Purify on windows)
  - Using a debugger (gdb, ...) helps with the other errors
  - Avoid errors using sensible data structures and built-in checking (assert)
  - Some systems in Crystal Space have complex debug tools written specifically for them (bugplug plugin)

Debugging: BugPlug

- BugPlug is a plugin that can transparently load in any Crystal Space application
- It listens for the mouse and keyboard and catches the ctrl-d key
- Can do various things like disable lighting, texture mapping, ...
- Can also communicate with built-in debuggers located in other modules (like visibility culler)

Debugging: Assert

- Crystal Space has a CS_ASSERT keyword that is only enabled in debug mode
- Can be used to include various validity tests throughout the code
- Makes code run slower (but only in debug mode) but you are sure that the validity tests will be right
- Basic classes like csArray (growing arrays) have lots of these CS_ASSERT calls to catch out-of-bounds access

The Engine

- Engine Plugins
- Sectors and Portals
- Mesh Objects
- Render Loops
- Visibility Determination
- Level of Detail
- Materials and Shaders
- Lighting Techniques
- Collision Detection and Physics
- Animation and Time

Engine: Plugins

- The engine itself is made out of several plugins:
  - Core engine
  - Visibility Culler
  - Mesh objects
  - Renderloops
- Engine communicates with renderer to render objects
- Renderer plugins:
  - 3D Renderer
  - 2D Canvas
  - Shader manager
  - Shader plugins

Engine: Sectors and Portals

- Basic concept in the engine is a sector:
  - Infinite area of space
  - Contains lights and mesh objects
  - Portals can connect different sectors
  - Mesh objects represent 3D geometry
- A portal is a polygon that points to another (or the same) sector
- Special portal effects are possible: mirrors, space warping
- Portals can move (they are in fact another mesh object)
- Portals have a great effect on level design: possibility to separate indoor and outdoor areas, ...

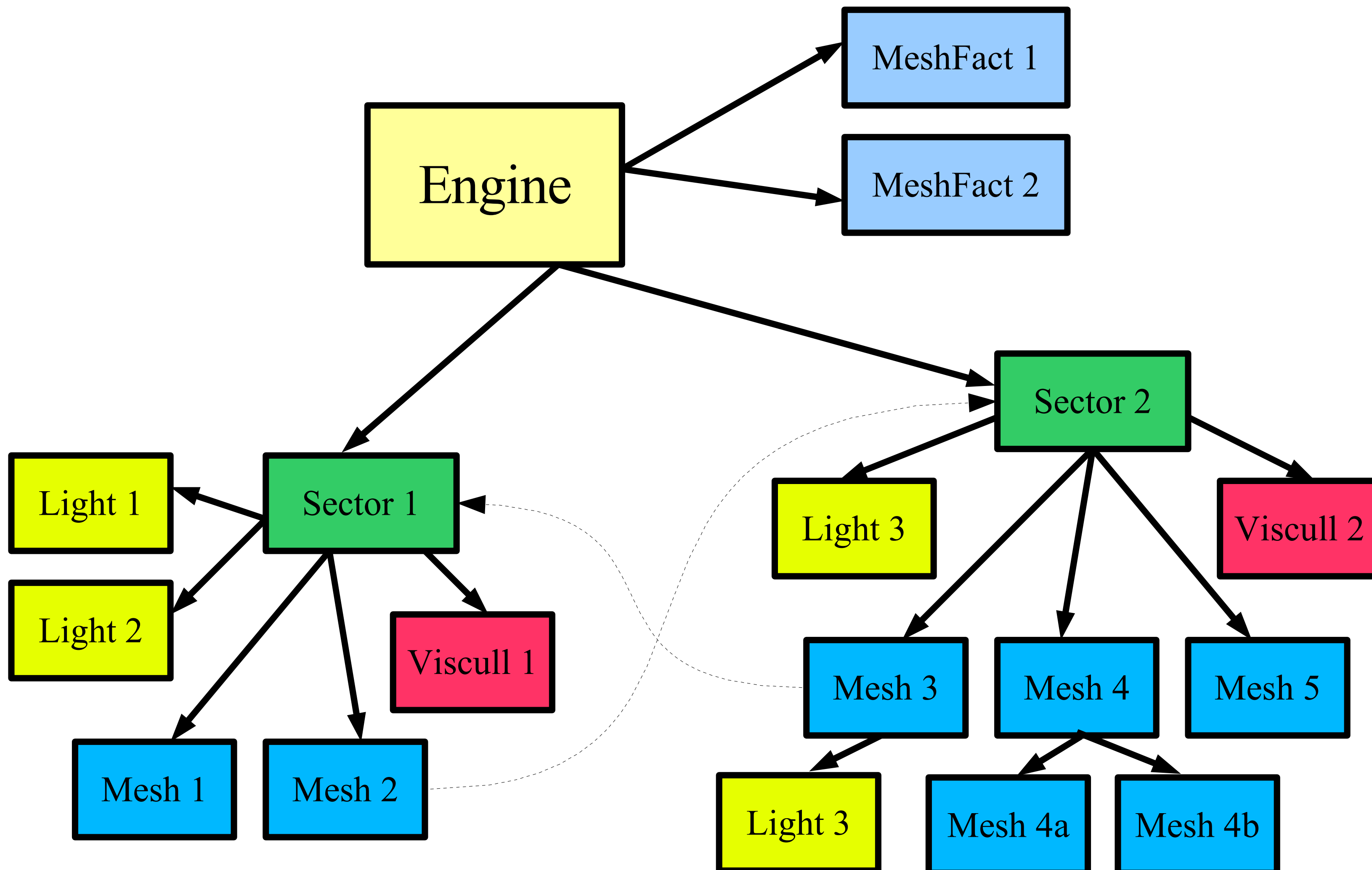
Engine: Mesh Objects

- Different types of Mesh Objects:
  - Triangle mesh object
  - Animated objects
  - Particle systems
  - Landscape engine
  - Portal container
  - Haze
  - ...
- Mesh objects can be put in a hierarchy
- Lights and camera can also be included in same hierarchy
- They can move, rotate, and move between sectors

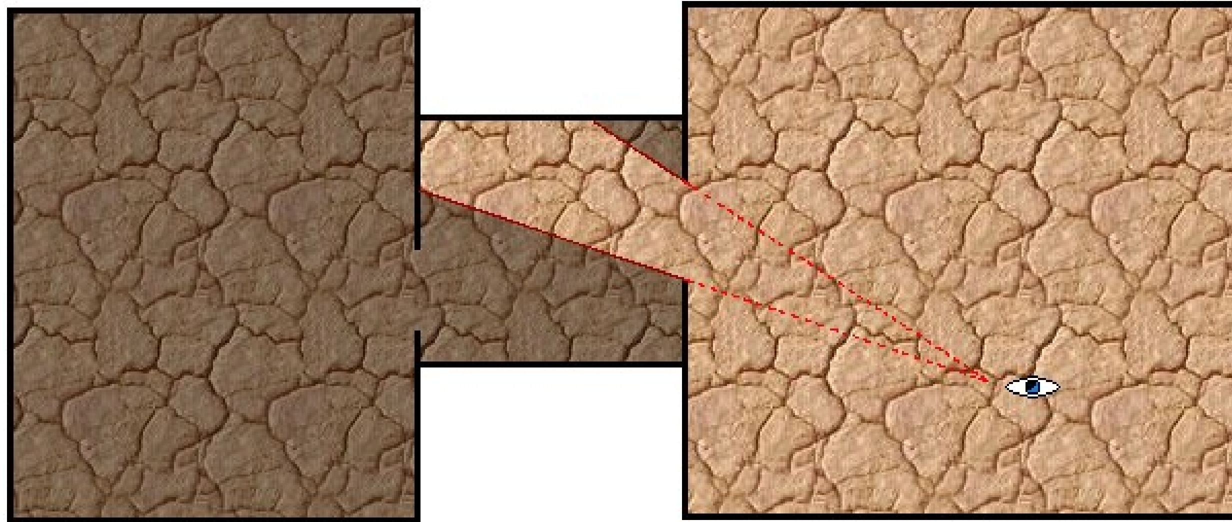
Engine: Mesh Factories

- Mesh objects represent all the geometry in the world
- A mesh factory is a template for a mesh object:
  - Contains the actual static geometry
  - Multiple mesh objects can be created from one mesh factory
  - Engine keeps a list of mesh factories

Engine: World Structure



Engine: Portal Example



Engine: Render Loops

- A renderloop gives the application developer fine control over how the engine should render objects
- A renderloop is made from different steps (plugins)
- A step defines other steps and/or a shadertype and various other attributes
- Several standard render loops are available:
  - Diffuse
  - Ambient
  - Stencil
  - Terrain
- Render priorities are another technique to give fine control to the game developer
  - You can tell the engine which objects are supposed to be rendered first

Engine: Render Loop Example

```
<params>
  <name>std_rloop_diffuse</name>
  <steps>
    <!-- ambient lighting -->
    <step plugin="crystalspace.renderloop.step.generic">
      <shadertype>ambient</shadertype>
      <zoffset>yes</zoffset>
      <zuse />
    </step>
    <step plugin="crystalspace.renderloop.step.lightiter">
      <steps>
        <!-- diffuse lighting -->
        <step plugin="crystalspace.renderloop.step.generic">
          <shadertype>diffuse</shadertype>
          <zoffset>no</zoffset>
          <ztest />
        </step>
      </steps>
    </step>
  </steps>
</params>
```

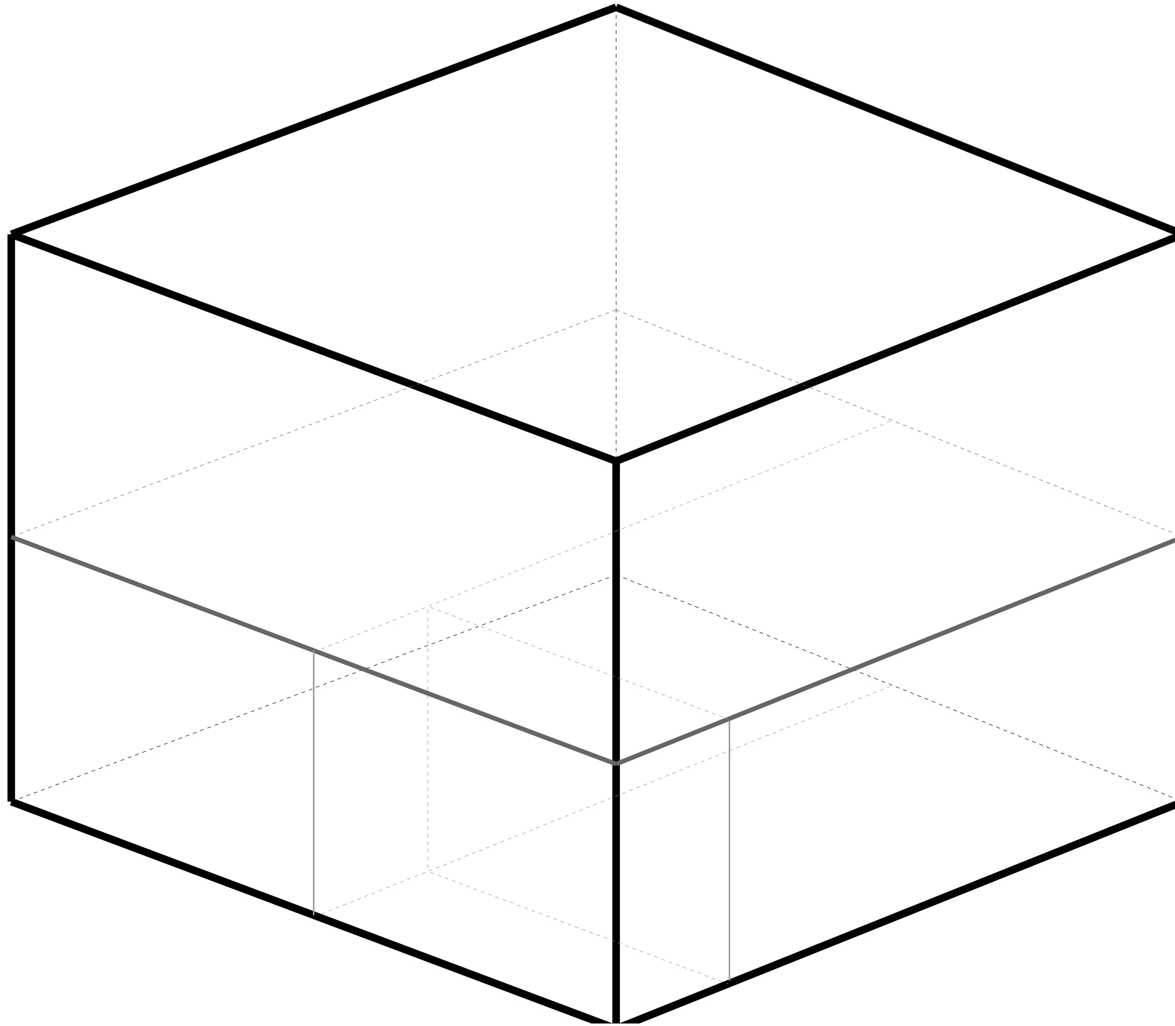
Engine: Visibility Intro

- The quest for speed:
  - Two conflicting goals:
    - Try to avoid rendering invisible (occluded) objects
    - Try to use as little CPU as possible (let the GPU do the work)
  - Visibility culling should be lightweight. Modern hardware is capable of rendering a lot of geometry so this should be kept in mind
  - Every sector has one culler (plugin)
  - Three cullers in Crystal Space:
    - Frustvis: only cull objects that are outside view frustum: useful for simple sectors or sectors that have lots of empty space
    - Dynavis: complex culler that tries to find out if objects occlude other geometry. Useful for heavy indoor areas or areas with many occluders
    - Pvsvis: not currently working. Useful for static geometry. Uses precalculated PVS (Potentially Visible Set)

Engine: Visibility KD-Tree

- Representing the geometry:
  - KD-Tree is used for Frustvis, Dynavis, and PVSvis
  - Hierarchical tree that divides space in a tree of boxes
  - Dynamic: objects move around and the tree adapts (not in case of PVSvis)
  - Adaptation happens in a 'lazy' way: avoid updating the tree unless that part of the tree is needed
  - First step of culling for Frustvis, Dynavis, and PVSvis is frustum culling (based on the hierarchy of the KD-Tree)

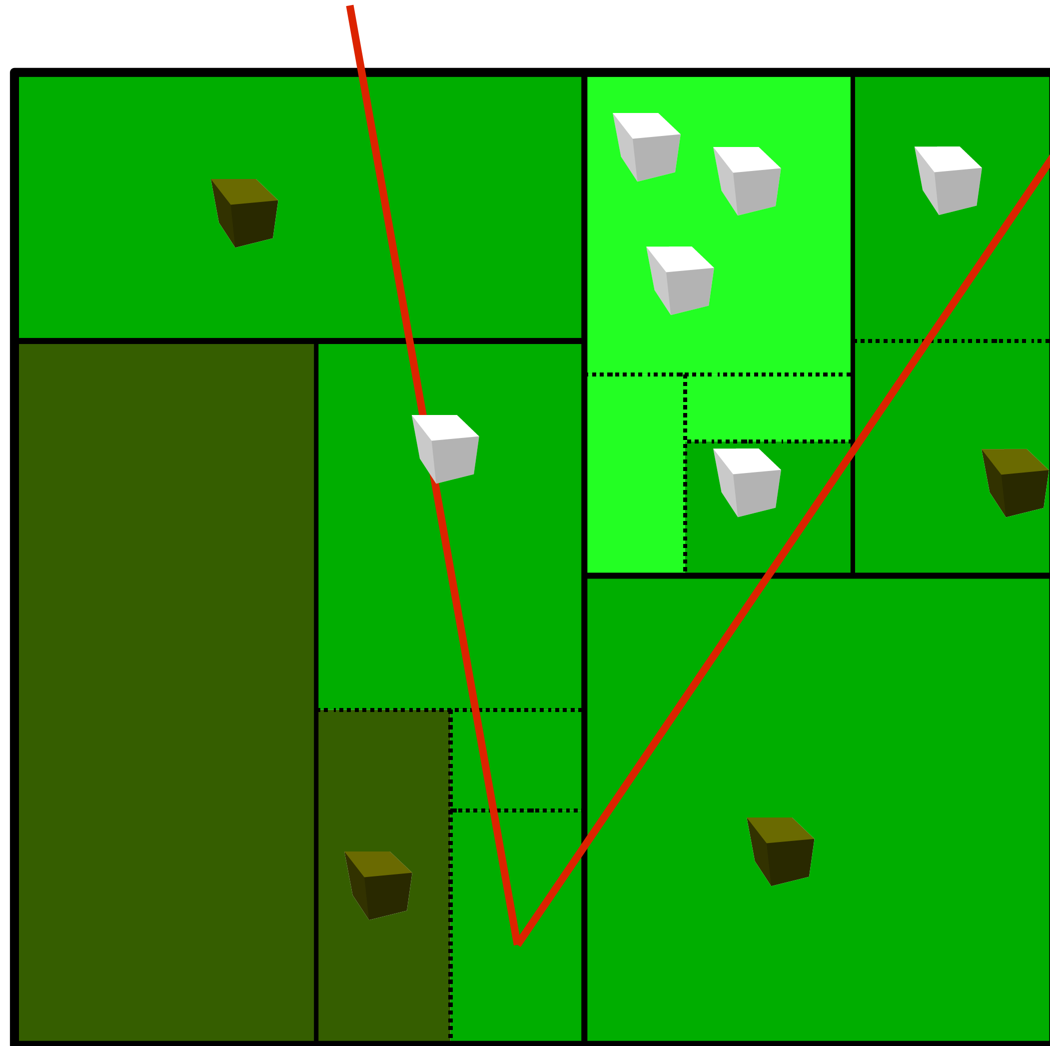
Engine: Visibility KD-Tree



Engine: Traversing the Tree

- Front to Back:
  - Dynavis traverses the KD-Tree in front-to-back order
  - Both Frustvis and PVSvis traverse the KD-Tree non-ordered
  - The traversal starts at the root node (biggest node) and then proceeds with the two children:
    - First the child containing the camera
    - Then the child not containing the camera
  - For every node we are in we recursively traverse by always taking the node closest to the camera first
  - This traversal is crucial for Dynavis. In this front-to-back traversal we roughly get all objects in front-to-back order. Every object is considered a potential occluder (can occlude other objects) and will potentially occlude objects that will come later in the traversal

Engine: Frustum Culling



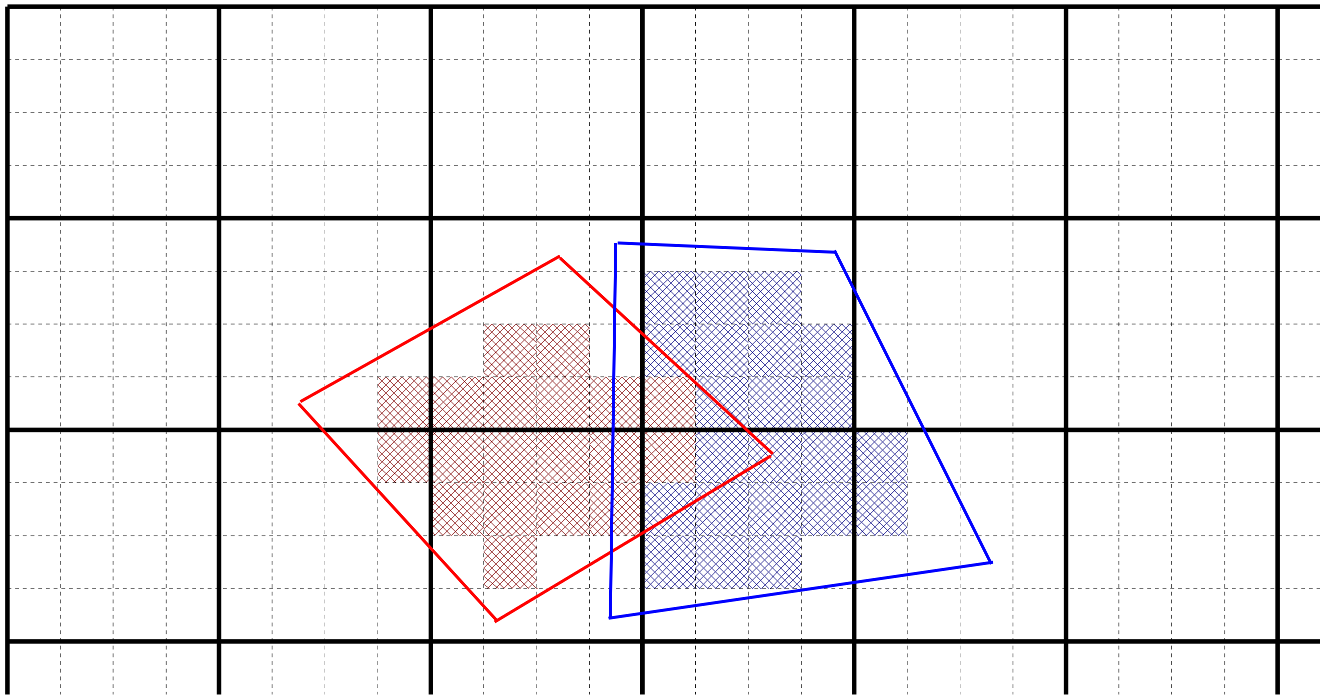
Engine: Dynavis Coverage

- How to test if objects are covered:
  - The basic concept in Dynavis is the coverage buffer
  - Works in screen space (requires expensive projection from geometry to screen)
  - Made from 32x32 tiles. Every pixel is represented by one bit
  - Due to the tiles lots of optimizations are possible (full tiles and empty tiles are marked)
  - Potential occluders are written to the coverage buffer by drawing their outline and doing a XOR-Sweep

Engine: Dynavis Depth Buffer

- How to test if objects are covered:
  - Associated with the coverage buffer Dynavis also keeps a depth buffer
  - Every 8x8 pixels (is block) has one depth value (so 4x4 depth values per tile)
  - The depth of one block is the Z value that is most far away from the camera (as opposed to normal Z-buffer)
    - Unless a polygon completely covers a block (optimization)
  - When updating the coverage buffer the depth buffer is only updated whenever the contents of the coverage buffer changes

Engine: Dynavis Coverage Buffer

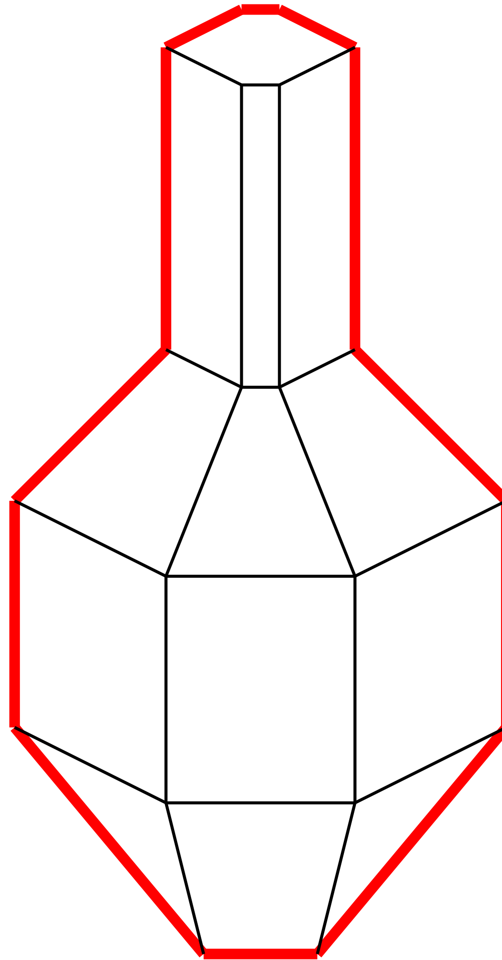


- Red polygon is in front of blue polygon
- Red polygon is inserted in coverage buffer first

Engine: Rendering Outlines

- Improving speed of coverage buffer:
  - Writing to the coverage and depth buffer is expensive
  - Doing this for every individual polygon would give most accurate culling results -> but high overhead
  - Instead we write the outline of the object to the coverage buffer
  - An outline is calculated in such a way that it can be reused even if the camera moves. The angle at which point the outline becomes invalid is calculated and used

Engine: Render Outlines



Engine: Test Rectangles

- Using the coverage buffer to test visibility:
  - To test if an object is visible we can:
    - Test every individual polygon and try to find one that is visible
    - Test the outline of the object
    - Test the bounding rectangle (in screen space)
  - Experimentally we found that testing the rectangle gives the best speed even though it is the least accurate test (i.e. objects can be marked as visible even if they are not)
  - We also test nodes against the coverage and depth buffer. The ability to cull away entire KD-tree nodes is one of the great savings of Dynavis

Engine: Write Queue

- Avoiding the expensive coverage buffer writes:
  - Writing to the coverage and depth buffer is expensive
  - Avoid this by delaying the write and storing the object until we need to test an object that intersects (in screen space) with the object we want to write
  - To test for intersection we test bounding rectangles
  - Delaying writes has two major advantages:
    - Avoids writing occluders that will never be used because there are no other objects behind them
    - By not writing occluders the depth buffer will have better values
  - The better values in the depth buffer make this one of the most important optimizations in Dynavis

Engine: History Culling

- Don't work unless you really have to:
  - Observation: an object that was visible this frame will most likely be visible next frame too
  - Use this fact to avoid testing visibility on visible objects for a random number of frames
  - Combined with the write queue this can mean that the write queue has to be flushed even less often

Engine: Visible Point Tracking

- Fast proof of visibility:
  - If an object is visible we only have to prove it is visible again next frame (ignoring history culling)
  - Find one visible point on the object and next frame test only that point
    - If that point is still visible then object is visible
    - If point is invisible we proceed with the bounding rectangle test
    - Testing visibility of a single point is very fast

Engine: Outline Splatting

- Try to use outlines that go beyond view plane:
  - The outline of an object cannot easily be used when the object goes partially behind the view plane
  - Nevertheless those objects are often very good occluders (floor of a building, walls next to the player, ...)
  - Solution: project the outline in a conservative way to the view plane

Engine: Dynavis and Portals

- One Dynavis instance is responsible for one sector
- If the Dynavis sector was entered through a portal:
  - Initial culling view frustum is restricted to portal area
  - Coverage buffer is filled outside portal polygon

Engine: PVS

- Potentially Visible Set
  - Precalculated data structure
  - Uses a hierarchical tree (KD-tree)
  - Every node contains a list of other nodes that are certainly invisible from that node
  - Depends on static geometry
  - Computationally expensive to precompute
  - At runtime the overhead is VERY low
  - This technique is very good for (mostly) static worlds

Engine: Level of Detail

- Visibility culling is one way to improve speed
  - Avoid rendering objects that are not visible
  - Not effective in open areas where nearly everything is visible
  - Computation of visibility is costly
- Level of detail (LOD) is one other way to improve speed
  - Render visible objects with lower detail (if object is far away for example)
  - Can improve speed even if all objects are visible
  - Relatively cheap in CPU power (depends on algorithm)

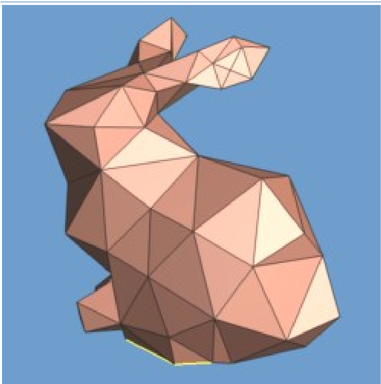
Engine: LOD Types

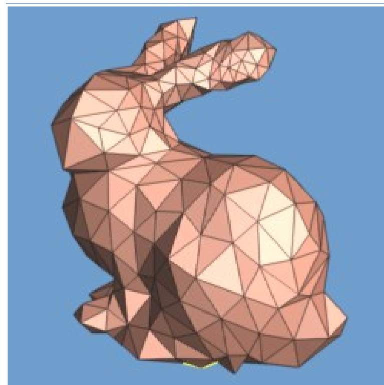
- LOD selection can be based on several factors:
  - Distance between camera and object
  - Hardware capabilities and speed
  - Desired framerate (user control)
  - Desired quality (user control)
- Different types of LOD
  - Progressive Mesh: triangle mesh data with a single material can be relatively easily reduced in detail by (for example) collapsing edges
  - Imposter LOD: when an object is far away, render the object on a texture and then render that texture instead
  - Static LOD: different versions of the same model
  - Terrain LOD: landscape engines typically have specific algorithms for LOD based on the heightfield data

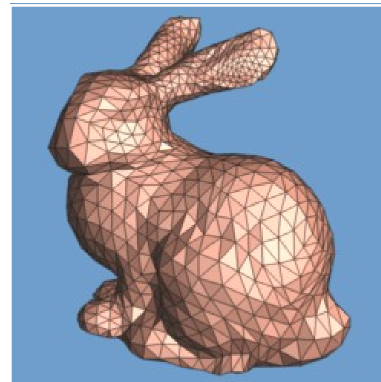
Engine: Progressive LOD

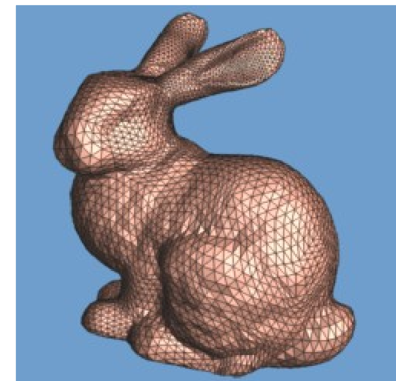
- Minimizing the number of triangles in a mesh:
  - Takes a triangle mesh as input and 'collapse edges': move the two vertices of the edge on each other (edge is removed)
  - Start from highest detail mesh
  - Find edge that, when collapsed, changes the visual appearance of the object the least
  - Remember that edge and then find the next edge
  - Result is a table from high detail to low detail that can also be played in reverse to form the mesh in the desired detail
  - Advantages: can produce any level of detail for the mesh in real-time, hardly any LOD popping
  - Disadvantages: if mesh animates then precalculated table may be sub-optimal. There are solutions to this but CS doesn't implement them yet

Engine: Progressive LOD









- Images taken from Hugues Hoppe (Microsoft Research)

Engine: Imposter LOD

- Replace an object with a single texture:
  - Render an object on a texture (procedural texture) and use that instead of object
  - Texture must be invalidated if:
    - Lighting changes
    - Object animates
    - Object rotates relative to the camera (some angle)
  - Depending on distance the above invalidation factors can be relaxed somewhat
  - Advantages: works on all types of objects, can even be used on multiple objects at the same time
  - Disadvantages: proc texture takes video card memory, updating a texture can be expensive, LOD popping if texture is updated

Engine: Imposter LOD (Portals)

- A special case of imposter LOD works on portals:
  - Replace portal contents with a texture
  - Invalidate when portal destination changes significantly

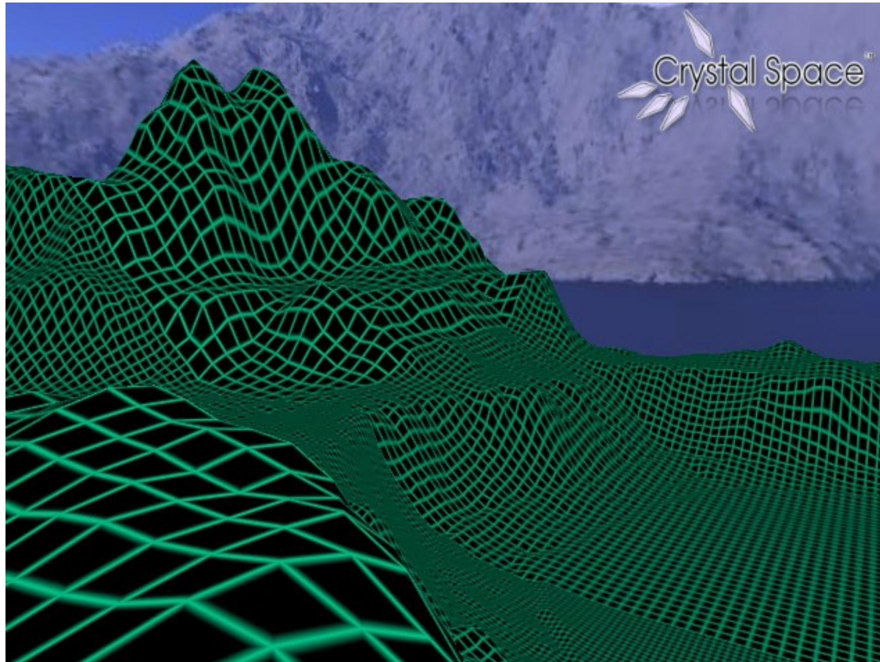
Engine: Static LOD

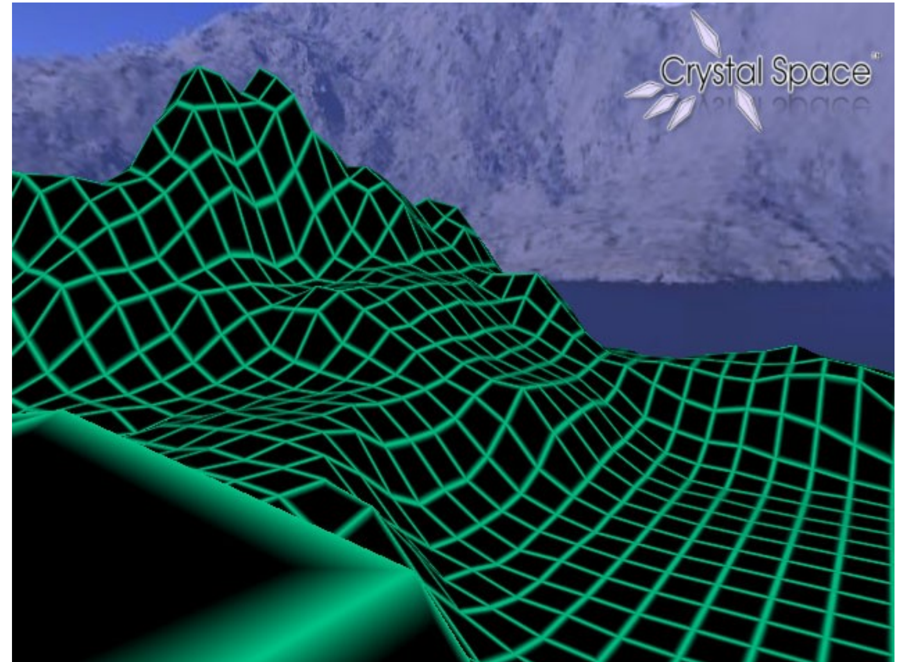
- Use different models with different detail:
  - Advantages: very cheap for the CPU (no work at all), lower detail objects can be made very well by artist
  - Disadvantages:
    - more work for the artist
    - LOD popping (visible disturbance when the model that is shown is changed)
    - more memory usage for the different model representations that have to be in memory

Engine: Terrain LOD

- Special case of simplification useful for heightmaps:
  - Starts with one block overlaying terrain (grid of 32x32 vertices)
  - Engine calculates distance between camera and block. If close enough block is divided in four new blocks
  - This happens up to a certain configurable level
  - Minimum seams between adjacent blocks because there can be only one level of LOD difference between two adjacent blocks
  - Makes a LOD transition at the edges of two adjacent blocks to help even more

Engine: Terrain LOD





Engine: Materials and Shaders

- Defining the surface attributes of an object:
  - Almost all mesh objects need 'materials' to define surface attributes
  - A material is defined from zero or more textures (2D images) and a shader
  - Shaders:
    - Different shader plugins: CG, ARB, fixed
    - Two main types of shaders: vertex and fragment shaders
    - A shader is a type of program that can run on the 3D card
    - A shader supports different techniques
    - Sorted by priority: highest priority technique that works on current hardware is selected. Using this technique you can make games that look very nice on high-end hardware but will still work on low-end hardware
    - A technique is made out of passes
    - Shaders can create special effects: bumpmapping, scattering, ambient, ...
  - Procedural textures

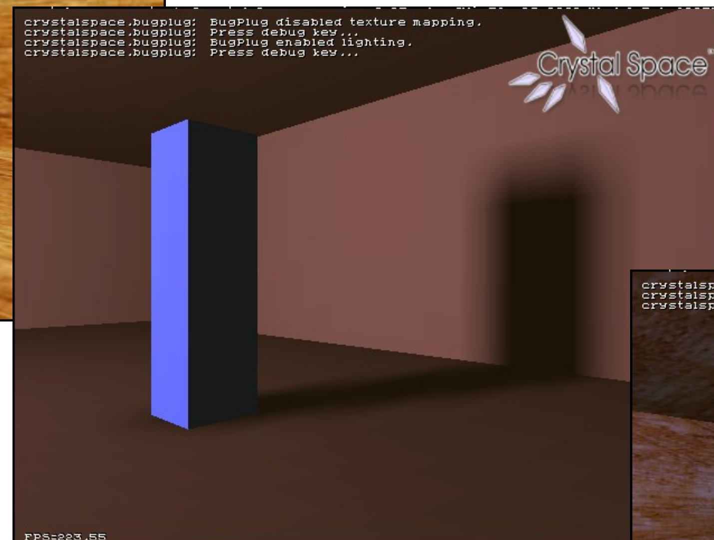
Engine: Lighting Techniques

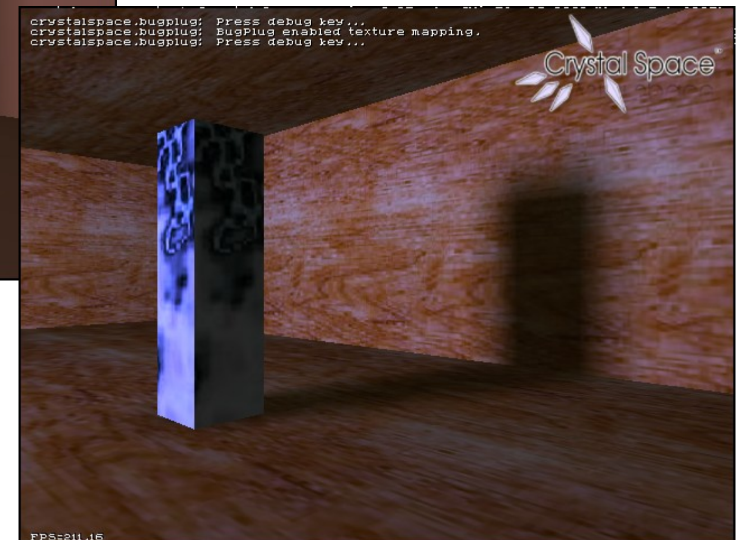
- Crystal Space supports different lighting techniques (through shaders and engine internals):
  - Lightmapping
  - Vertex lighting (gouraud shading)
  - Stencil based shadows
  - Comparison
- Halos and lens-flares

Engine: Lightmapping

- Soft shadows with lightmaps:
  - Lightmap is a low resolution texture that is blended with the material on the polygon to give lighting effect (by default one 'lumel' for 16x16 'texels')
    - This results in 'soft' shadows because the shadow information is only calculated every 16x16 texels
  - Lighting and shadows are calculated in a precalculation pass and then stored on disk. Calculating lighting can take hours for big maps
  - Updating lightmaps at runtime is not possible in practice due to long calculation times. But:
    - In addition to lightmaps there are also pseudo-dynamic shadowmaps. There is a shadowmap for every polygon/pseudo-dynamic light combination. Pseudo-dynamic lights can change color and intensity. The lightmap and shadowmaps are merged at runtime
    - Dynamic lights also affect the final resulting lightmap. Quality is not high (no shadows)

Engine: Lightmapping





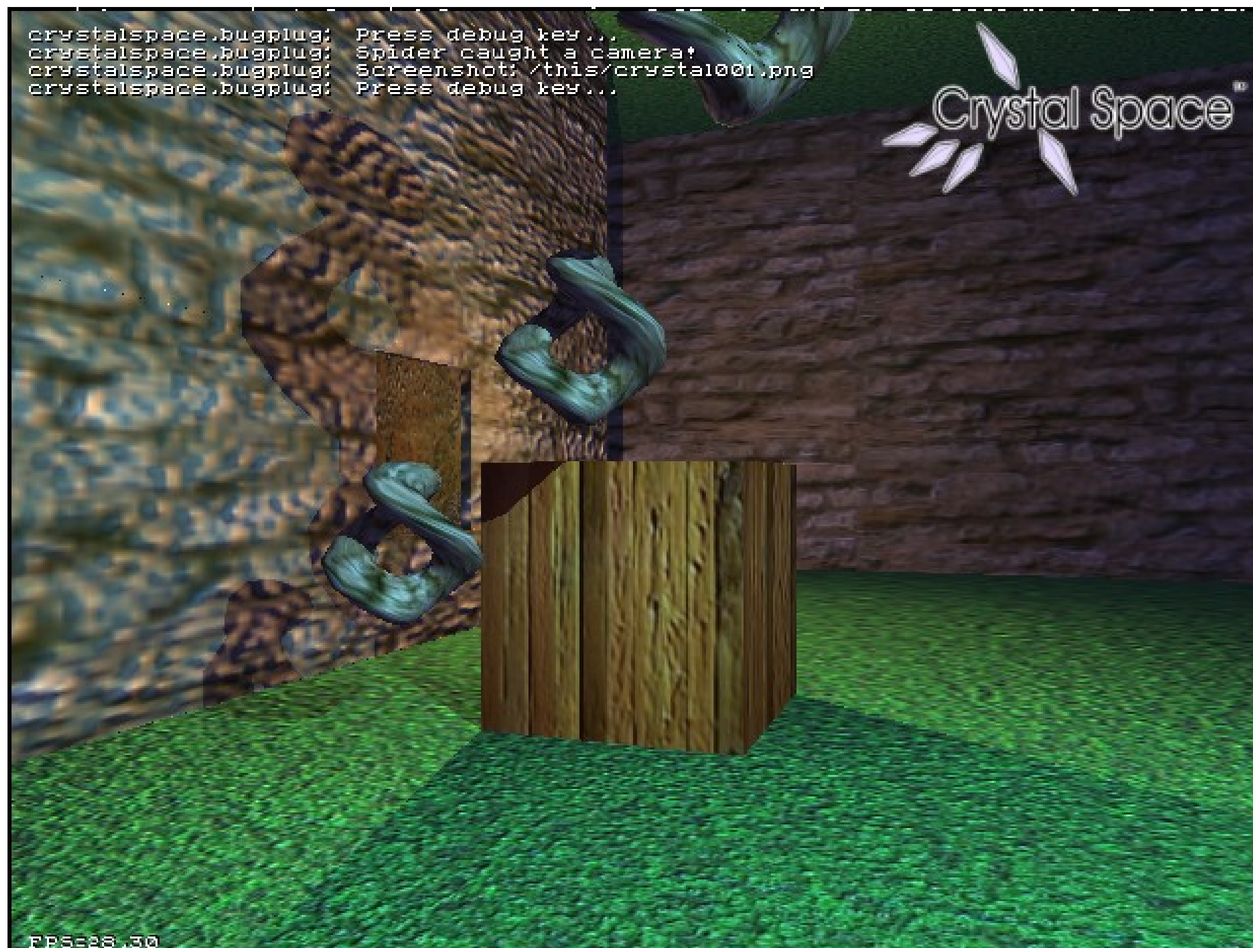
Engine: Vertex Lighting

- Soft shadows using vertex lighting:
  - There is one light value per vertex of an object which is updated by all the lights that hit affect the object
  - Updating of lighting is fully dynamic
  - Shadows are possible but accuracy depends on the amount of vertices an object has
    - Also soft shadows

Engine: Stencil Shadows

- Hard shadows using stencil hardware
  - The 3D scene is rendered again from the perspective of each light
  - Creates sharp and accurate shadows

Engine: Stencil Shadows



Engine: Lighting Compared

- Lightmapped lighting:
  - Pros:
    - Very fast at runtime: no runtime cost at all
    - Speed independent on the number of lights
    - Nice looking soft shadows
    - Accuracy of shadows is independent of polygon size
    - Radiosity is possible
  - Cons:
    - Slow to precalculate
    - Hard to update at runtime
    - Shadows not very accurate
    - Memory usage of lightmaps can be high

- Stencil lighting:
  - Pros:
    - Sharp accurate shadows
    - Dynamic shadows: immediately updated when objects move
  - Cons:
    - Slows down with large maps and high number of lights
    - Visual conflict with soft shadows
    - Requires hardware support (stencil buffer)

- Vertex lighting:
  - Pros:
    - Reasonably fast at runtime: for static lights can be precalculated
    - Nice looking soft shadows
    - Easy to update at runtime
  - Cons:
    - Accuracy of shadows depends on tesselation factor of object

Engine: Collision Detection/Physics

- Collision detection systems:
  - OPCODE
  - Works by taking the geometry and creating a hierarchical tree of oriented bounding boxes for it
  - Calculation of intersection happens on the biggest box first
- Physics system:
  - ODE
  - Bullet
  - Simulation of real-life physics

Engine: Animation and Time

- Keeping animation independent of framerate
- Keyframe based animation
- Skeletal animation
- Hierarchical animation
- Particle system animation
- Other types of animation

Engine: Time Independence

- Framerate and animation:
  - Speed of animation should be the same on a fast computer and on a slow computer
  - Speed is expressed in framerate (FPS)
  - To correct animation so that it runs at the same speed regardless of FPS the 'elapsed time' is used
  - Elapsed time is the difference between the time of this frame and the previous frame. It is the best estimate of how long the next frame will take
  - All animation in Crystal Space uses that elapsed time to control how much of the animation has been done

Engine: Keyframe Animation

- Fixed animations for triangle meshes:
  - A 3D model has a single triangle mesh
  - Different sets of vertices, every set defines a 'frame'
  - By switching frames there is the feeling of animation
  - Frame based animation is limited. The model can only animate through the frames. Other animations are not possible
  - Actions are groups of frames (for example: 'run', 'stand', 'walk', ...)
  - Elapsed time is used to control when to switch frames. In combination with frame interpolation smooth switching can be done even with few frames

Engine: Skeletal Animation

- Flexible animations for triangle meshes:
  - Skeletal (or bone) animation breaks the model in different parts (bones)
  - But it stays a single model
  - The parts are positioned relative to the parent
  - Vertices can belong to several bones (vertex weighting)
  - By modifying the transforms between bones lots of different motions are possible
  - Skeletal animated models use less memory
  - Animating them is a bit harder in general (hard to control the exact parameters of the transform)

Engine: Hierarchical Animation

- Hierarchical animations for objects of mixed type:
  - Hierarchical animation is a similar to skeletal animation
  - Difference is that the hierarchy is made from truly independent models
  - More flexible in the sense that you can easily attach some object to another object (like attach a sword to an actor)
  - Vertex weighting is not possible though

Engine: Particle Animation

- Animation of single particles:
  - Particle animation is another example of animation that needs to be independent of FPS
  - Particle systems use (semi-)physics to control position of the individual particles

Engine: Other Animation

- Crystal Space has a sequence manager which is a simple scripter that you can use to:
  - Move objects
  - Move and change light parameters
  - Change textures (texture animation)
  - ...
- Texture animation:
  - Switching textures at runtime
  - Animated gifs or mng files
  - Procedural textures
  - Movie support

Renderer: Introduction

- The engine, renderloops, mesh objects and visibility cullers form one side of the Crystal Space framework
- The renderers, canvas plugins, and shader plugins form the other side
  - Canvas: is responsible for opening the window (or initializing the screen), the 2D area on which to render, and some 2D operations
  - Renderer: does all the 3D operations. Uses the canvas as the render area and the shaders to get various effects
  - Shaders: responsible to control how materials are rendered

Renderer: Canvas Plugins

- Different canvases for different window systems and operating systems:
  - Windows DirectDraw (for software renderer)
  - Windows WGL (for OpenGL renderer)
  - Linux X11 (software)
  - Linux SVGALIB (software)
  - Linux GLX (OpenGL)
  - Macintosh Cocoa (software)
  - Macintosh GLOSX (OpenGL)
  - Null canvas (no display)
  - Asciiart (ascii char display)
  - ...

Renderer: Different Renderers

- The renderer is responsible for 3D operations:
  - Drawing polygon meshes
  - Controlling stencil hardware
  - Managing textures
  - Managing fog
  - ...
- Different renderers:
  - OpenGL
  - Software
  - Null (for server usage)
  - ...

Renderer: Shaders

- Shaders control the surface attributes of mesh objects
- Different types of shaders:
  - ARB (using common ARB OpenGL extensions)
  - Fixed (using standard OpenGL calls)
  - CG (using nVidia CG language)
  - ...
- Can be used to control various aspects of a surface like bumpmapping, blended materials, scattering, ...

Various Plugins

- Console plugins:
  - Output consoles are responsible for displaying status information from the game/application
  - Input consoles can accept in-game commands
- Joystick input
- Font plugins:
  - Csfont: fixed size fonts
  - FreeFont: variable size fonts

Various Plugins

- Scripting plugins:
  - Java
  - Perl
  - Python
- XML Parsing plugins
- Sound plugins:
  - Sound loader plugins: WAV and OGG
  - Sound driver plugins: OSS, ALSA, WaveOut, CoreAudio, ...
  - Sound renderer plugins: Software and OpenAL

Various Plugins

- BugPlug:
  - Debugging plugin
- MovieRecorder:
  - Can record a movie from any Crystal Space program while it is running
- Sequence Manager:
  - Manages time based operations
  - Allows for adding simple animations/interactions to a map file
- Reporter:
  - Standard system so that any plugin can notify the user that something is wrong

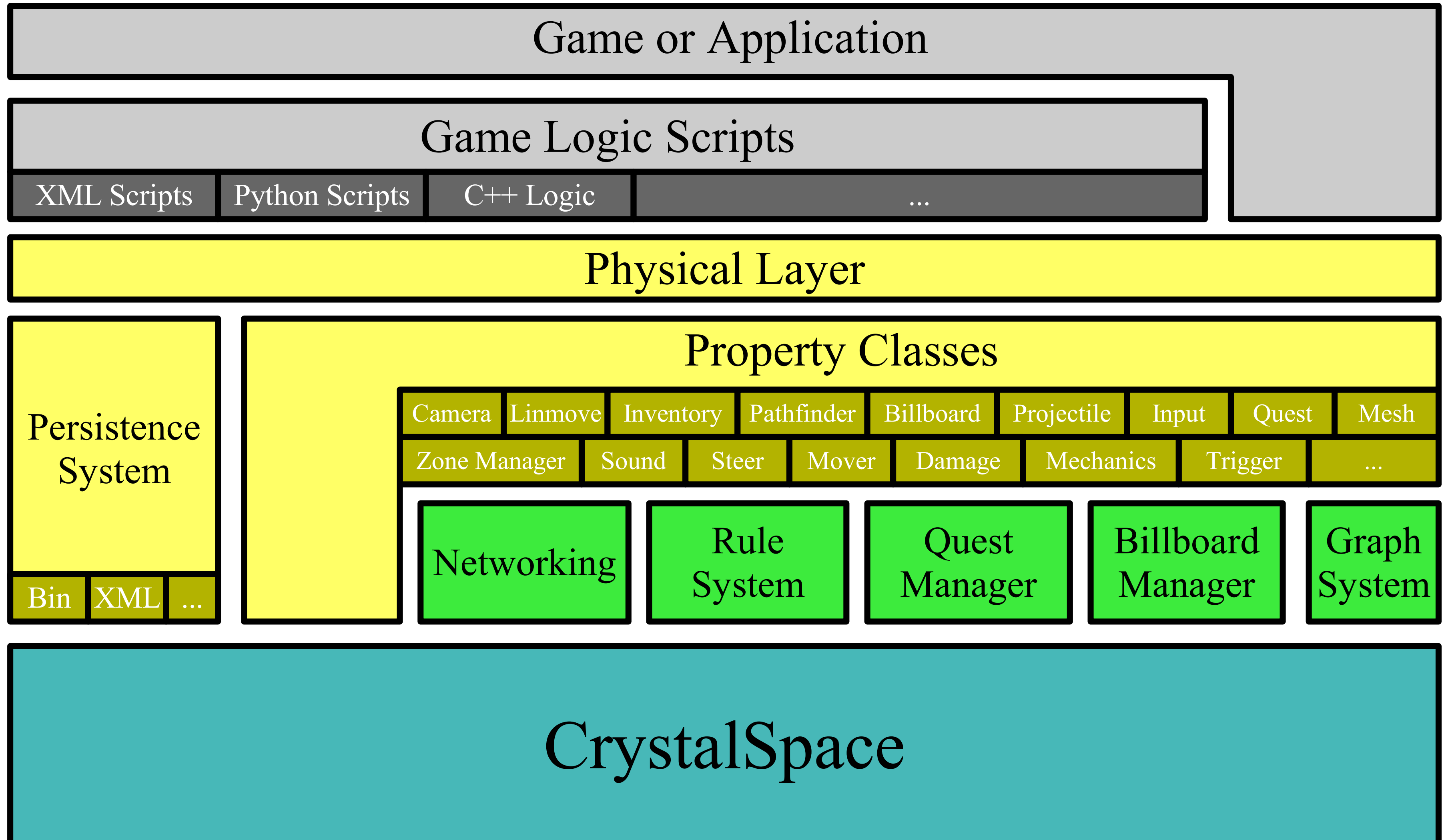
Various Plugins

- CEGUI:
  - Built-in window system based on CEGUI
  - Can be used for in-game menus
  - Integrates well with 3D view: 3D view in window, windows on top of 3D view
- VFS:
  - Virtual file system
  - Reads on regular file system and in zip archives
- ...

Crystal Entity Layer

- Crystal Entity Layer
  - Separate project from Crystal Space
  - Handles game entity concepts
  - Physical Layer:
    - Entity (like actor, monster, chest, ...)
    - Property classes control what an entity can do
    - Persistence (saving and loading game state)
  - Game Logic:
    - A game logic 'script' is also a property class but implemented by the game developer
    - Can use several languages: C++, Python, XML, ...

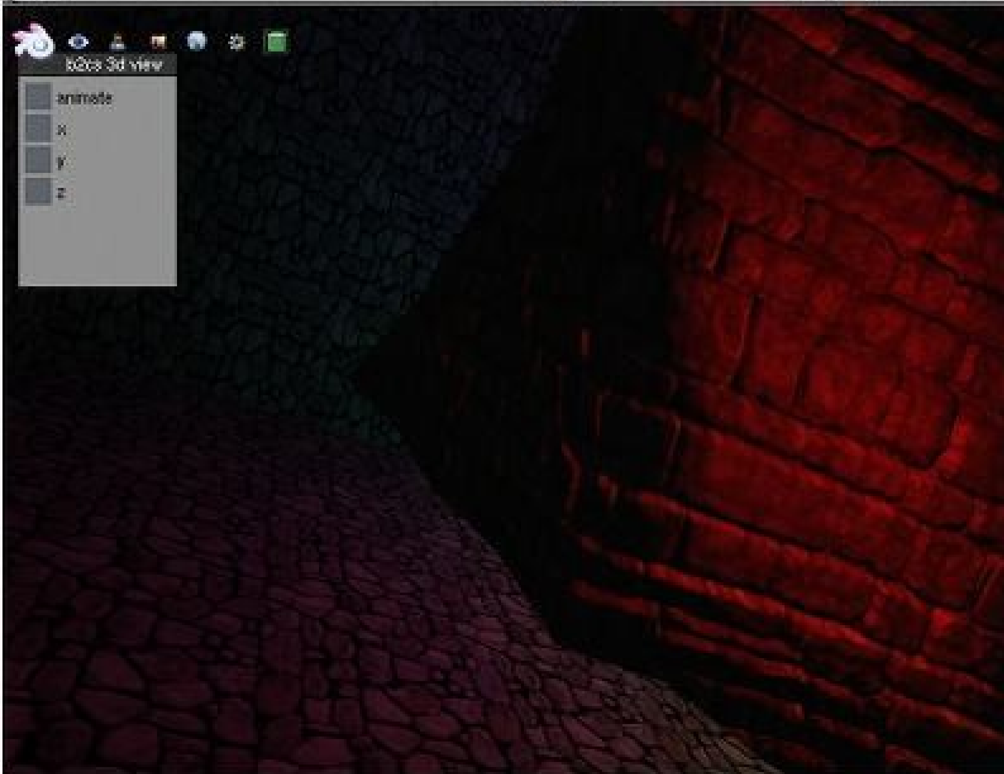
CEL: Architecture Overview



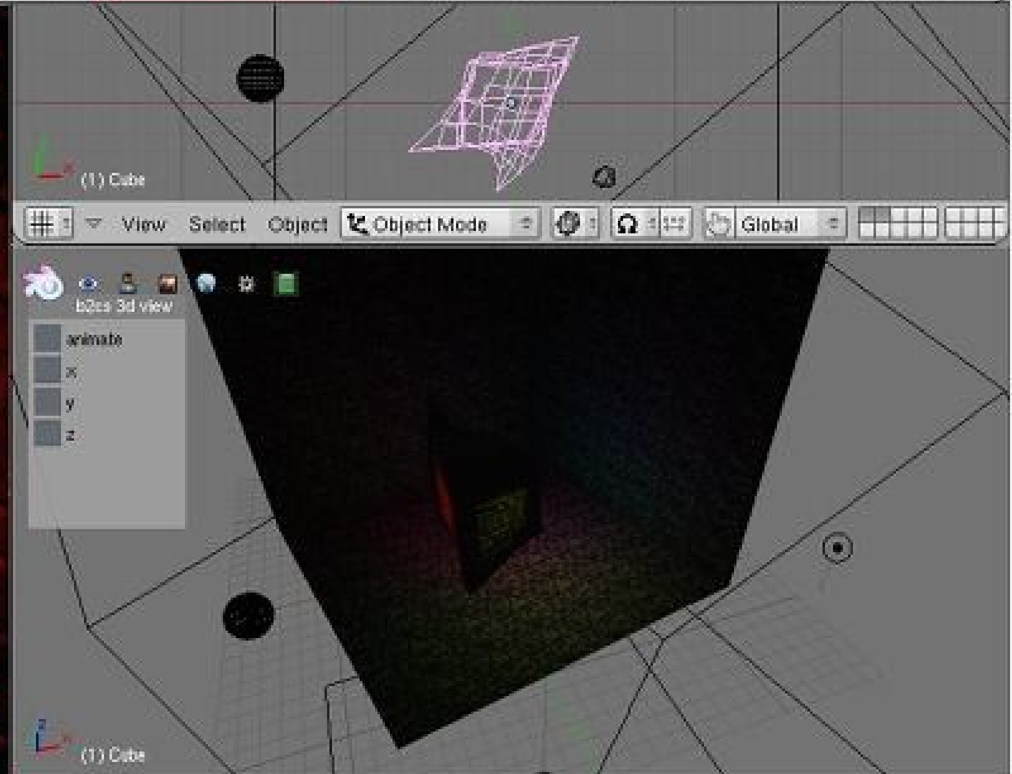
Apricot

- Open Game Project run by Blender Institute and Crystal Space
  - For six months six people will work hard to create a cartoon based game using Blender and Crystal Space
  - Project started beginning of February 2008
- Targets:
  - Streamline the game creation pipeline using Blender and Crystal Space (and in the future other engines as well)
  - Create a fun and commercial quality game using only Open Source tools
  - Create a community based on that game so that the project doesn't end after the six months have passed
  - Provide a tutorial and documentation for game developers
- Preorder the DVD at: http://apricot.blender.org

Blender [/home/caedes/CRYSTALSPACE/examples/materials/parallaxtest.blend]
File Add Timeline Game Render Help SR:2-Model SCE:Scene www.blender.org 245.14 Ve:1104 | Fa:1127 | Ob:7-1 | La:2 | Mem:2.28M (0.81M) | Time: | Cube



b2cs 3d view
animate
x
y
z



(1) Cube
View Select Object Object Mode Global
b2cs 3d view
animate
x
y
z
(1) Cube

View Select Object Object Mode Global
View Select Object Object Mode Global
Link to Object
MA:Material 2 X F Nodes
ME:Cube OB ME 1 Mat 1
Render Pipeline
Halo ZTransp Zoffs: 0.00
Full Osa Wire Strands ZInvert
VCol Light VCol Paint TexFace Shadeless
No Mist Env Shad A 1.000
Col R 0.756
Spe G 0.756
Mir B 0.756
Subsurface Scattering Custom
Scale: 0.100
Radius R 1.000
Radius G 1.000
Radius B 1.000
IOR: 1.300
Error: 0.050
Col 1.000
Tex 0.000
Front 1.000
Back 1.000
Lambert
CookTo
Panels 1
XChat: caedes @ caedes@delirium Perendev is Tool Blender [/home/c Escritorio 1 Escritorio 2 01:55 14-02-08

The End