Chapter 10 - Graphics, DRAFT 1
Jefferson Technical Design

Eric Bowman, October 28, 1997

10. Graphics

This chapter describes the graphics subsystem in Jefferson.

The goals for the graphics subsystem are:

- 20 fps or better "mixed-mode" rendering of terrain, z-buffered sprites, and polygons.
- Rendering effects that scale to the given capabilities of the user's machine.
- Different render schemes for different hardware configurations as needed (software, 3Dfx, etc.).
- Dynamic lighting of polygons and z-buffered sprites

TBD: What FPS would we settle for?

TBD: What is our target platform? Minimum platform?

10.1 Hybrid System

Jefferson mixes polygons and sprites to form a 3D, non-perspective, tile-based world.

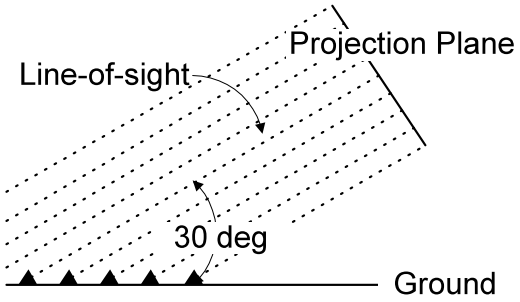
The 2D graphics consist primarily of sprites with z-buffers, rendered in 3D Studio MAX, and z-blitted in the game. The 3D graphics consist of texture mapped polygonal figures, (people and possibly pets), modeled and animated in 3D Studio MAX.

10.1.1 Coordinate Systems

Both tile and world coordinates are used by the graphics system. Chapter 2 gives an overview of the coordinate systems.

10.1.2 "Iso View"

The so-called "isometric view" is generated through a non-perspective (dimetric) projection from 3D view coordinates onto a projection plane representing the screen. It is a non-perspective projection onto a viewing plane oriented so that the viewing angle is 30°.



10.1.3 Video Resolution

The game will run at 640x480 pixels resolution, in "hicolor" mode (16 bits per pixel.)

TBD: The game will run also at 800x600 pixels resolution, if the hardware can support our requirements.

10.1.4 DirectX

The game will require DirectX 5, or (TBD) DirectX 6 (due to ship in Q298).

10.1.4.1 DirectDraw Modes

The game will run full-screen exclusive mode.

TBD: will it also run in windowed mode? If so, only for debugging, or also the released version?

10.1.4.2 Direct3D Devices

The game will run with either the Direct3D ramp software renderer (HEL device) if no 3D accelerator is present.

TBD: The game will use the MMX software renderer (HEL device) on an MMX machine.

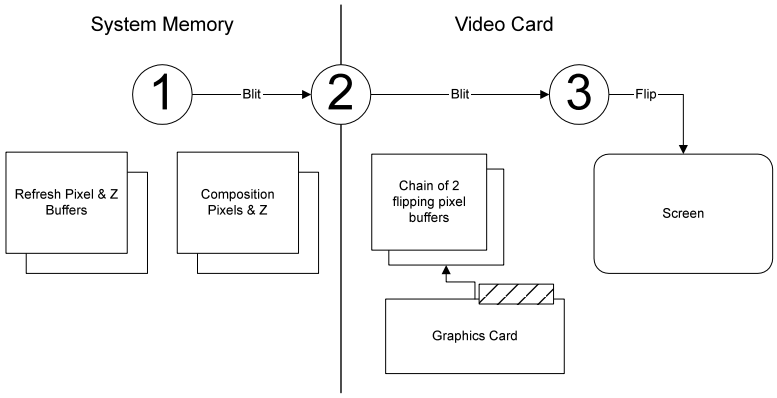
TBD: The game will use 3D hardware if it available (HAL device). Will we ever use the HEL even if a HAL is present?

TBD: Could we get significantly better performance in software using a custom software rasterizer?

10.1.5 Buffers

When using a software renderer in fullscreen mode, the game will have:

- 2 Flipping buffers on the graphics card.
- "Composition" pixel and z buffers in system memory.
- "Refresh" pixel and z buffers in system memory



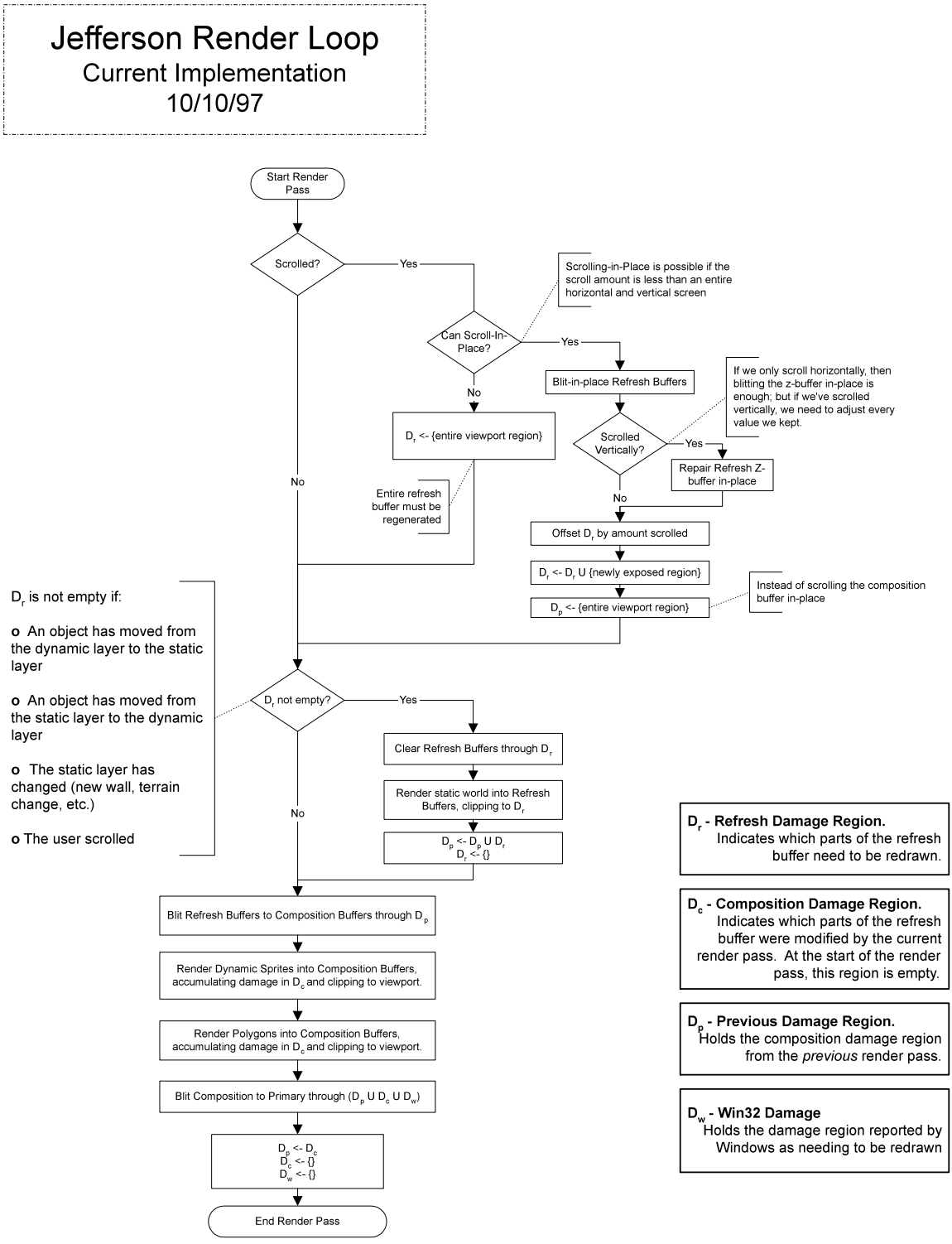
The basic render cycle for fullscreen without 3D hardware acceleration (see figure above) is:

1) Blit refresh buffers to composition buffers to "clear" them.
2) After rendering sprites and polygons, blit composition pixel buffer to on-card backbuffer.
3) Flip.

In windowed mode, on-card flipping buffers are not required; step 2 above will be to blit from the composition buffer directly to the primary display surface.

10.1.5.1 Clearing and Updating

The render loop clears and updates according to accumulated "damage" in various damage lists. Specifically, there are 3 damage lists, one for damage to the refresh buffers (which gets added when it changes; see below figure), and two for the composition buffers, for previous and current damage.



10.1.5.3 Scrolling

Scrolling uses the damage lists to minimize redraw. If the screen is only partially scrolled, the refresh buffers are scrolled-in-place, the refresh z-buffer is repaired in-place, and the freshly exposed regions are rendered. Then the entire refresh buffer is blitted to the composition buffer, and all the dynamic objects are redrawn.

TBD: How to make scrolling faster. We may use refresh buffers that are somewhat larger than the composition buffer to minimize the amount of refresh buffer that must be updated in real-time while scrolling.

10.1.5.4 Windows and Panes

Jefferson uses a single viewport for 2D and 3D graphics.

TBD: Whether we will use a DDD viewport for the tool panel when we go fullscreen.

10.1.6 Colors

Jefferson will only run in hi-color (16 bits per pixel) mode. It will support 555 and 565 pixel formats.

TBD: Whether we need to support 655 and 556 as well.

10.1.6.1 Per-Sprite Palettes

Currently there is a single 256-color primary palette shared by all sprites. There is also a single "sprite brite" palette generated from the primary palette that is used to draw highlighted sprites. These will be replaced by per-sprite palettes and palette sharing.

Each sprite will have a palette GUID. The GUID will be resolved to a palette object through a palette manager. Typically the sprite's palette GUID will refer to either a standard palette, or to a palette within the object in which the sprite is embedded. Typically an object will have a single palette to share among all its sprite, though this scheme allows us to get fancy later if we have to.

The palette will be stored in a resource as 256 24-bit RGB values. At runtime, a 256-entry color look-up table (CLUT) of 16 bits per entry will be generated from the 24-bit RGB values to match the particular video card's pixel structure (555 or 565).

The palette object may contain several palettes, including:

- the standard palette
- a "sprite-brite" palette used when highlighting a sprite
- other palettes as needed (night, various degrees of shading, etc.)

Note: The per-sprite palettes are different from DirectDraw palettes, and are only used by our software blitters.

TBD: These palette will probably be generated dynamically from the standard palette, though we may want an artist to be able to specify it if necessary. Thus a palette resources should contain a list of palettes, and an identification scheme to specify what each palette is (standard, sprite-brite, night, etc.)

TBD: How to generate the GUIDs, and how many bits is sufficient. 32 bits is probably more than adequate.

10.1.6.2 Palette Sharing

The palette manager will facilitate palette sharing. In particular, sprite objects with similar color schemes may be able to share on of the standard palettes. The more palettes there are (standard, sprite-brite, etc.), the more important

TBD: Whether an individual sprite's palette will be modified on-the-fly, in which case sharing-by-value won't work. A palette object will probably have a Clone member function so a modifiable copy can be created and changed without affecting other sprites which share the palette.

10.1.6.3 Terrain Colors

TBD: Everything about terrain.

Terrain probably will be drawn either as polygons using DDD, or by a special-purpose terrain renderer, optimized for non-perspective and to handle sheer.

10.1.6.4 Textures

Textures are stored as 8-bit, palettized BMP files. The DDD runtime dynamically converts textures to the "preferred bit" depth, which is 8 bpp unless 16 bpp is supported. If neither is supported, the texture format whose bits per pixel is closest to the screen depth is chosen.

10.1.6.5 Texture Palettes

At present, each texture has its own palette. This is wasteful, and might be slow on some 3D accelerator hardware.

TBD: A scheme for sharing palettes among texture objects.

10.2 2D Graphics

10.2.1 Sprites and Z-Sprites

Sprites are run-length encoded (RLE) 8-bit per pixel bitmaps. They are stored on-disk and in-memory in RLE form. Currently, A sprite is decompressed each time it is drawn.

Each sprite is associated with a palette; the decoded pixel data is index into the palette to convert it to the particular device's 16-bit pixel format

TBD: Whether decompressing sprites every frame is a significant bottleneck.

10.2.1.1 Z Sprites

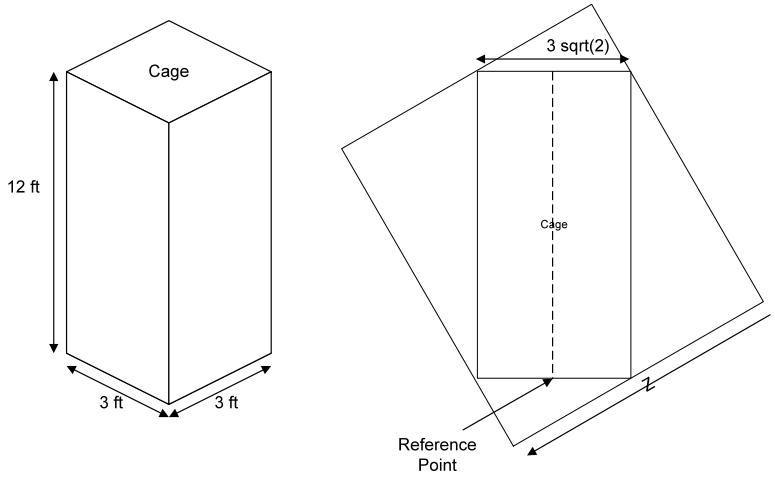
Z sprites are sprites that include a compressed z-buffer in addition to pixel data.

The z-buffer data is encoded at 8 bits per pixel. Each z value is the encoded distance, along the camera's z-axis, of the difference between the z-value at that pixel and the z-value of the bottom-center of the tile.

When the sprite is decompressed, the z-pixel is used to index into a lookup table that converts the z value to the correct value for the current z-buffer.

Z sprites are generated in 3DStudio MAX using a custom utility plug-in. An artist creates a 3D object in MAX that fits on a single tile (3 units by 3 units, where 1 unit = 1 foot) "cage." Then the plug-in renders the objects, and exports the pixel and z data.

The figure below shows the cage from the camera's perspective, and a side view of the cage showing the extent of z values that can be encoded in a given sprite at the current sprite z-pixel resolution.



"The Cage" and the Z Dynamic Range

TBD: The plug-in only approximates the game's projection matrix...should we investigate whether specifying the precise matrix is possible?

TBD: MAX often draws pixels for which there is no corresponding z-value. Shall we attempt to patch up these holes? (There are obvious holes, for instance, in the smallest rendering of "DinChair1".)

TBD: Do we want MAX to render all the zooms, or just a single zoom, and have the artists use Photoshop to reduce the image?

10.2.1.2 RLE Compression

Of 1257 sprites (at present), 210 are compressed to a size larger than an 8-bit uncompressed version of the same sprite.

TBD: Will we use a different compression scheme for sprites that get anti-compressed?

10.2.1.3 Decompression & Blitting

Blitter objects are used to decode RLE-compressed sprites, process the decoded data (z-compares, alpha blending, lookup table conversions, etc.) and write pixels and z values to buffers.

The current blitter code base is good for experimenting with different pixel pipelines, but is slow. TBD: whether attempting to optimize it is worth it, or whether we should abandon them entirely.

Our current inventory of blitters consists of

- cStandardBlitter16 - Blits pixels, no Z. Presently used to draw floor tiles. Soon to be obsolete.
- cStandardZBlitter16 - Blits pixel "decals" with Z, writing Z without doing a Z-test. Presently used to draw thought balloons. TBD: Whether to make this blitter read Z, or to draw thought balloons the same way wall sprites will drawn (computing z values in real-time).
- cZReadSpriteBlitter - Blits pixels and Z; does a <= Z compare on each pixel. This is currently the workhorse blitter.
- cZHiliteReadSpriteBlitter - Same as cZReadSpriteBlitter, but computes and uses a special "sprite brite" palette. This blitter will be obsolete once per-sprite palette sharing is in place.
- "cZBlendReadSpriteBlitter" - Experimental alpha-blending blitter.

TBD: What pixel effects will we end up doing, and how many separate inner loops do we need coded to accomplish them.

10.2.1.4 Sprite Symmetry

TBD: We will flip certain symmetric z-sprites decrease their memory footprints. Which sprites are flipped will be decided on an object-by-object basis, pending an object list.

TBD: Who decides which objects are to be flipped?

10.2.1.5 Composite Sprites

Most "Objects" (see Chapter ???) use composite sprites to facilitate animation. For instance, the shower object has four z-sprites: shower (no door), and three door sprites, one for "closed," "half-open," and "open." When the object is drawn the shower sprite and one of the door sprites are blitted with the "cZReadSpriteBlitter."

10.2.4 Terrain

Terrain is currently 2D, and has no z-buffer.

TBD: How terrain will be done.

10.2.5 Static and Dynamic Layers

Rendering is split in to two phases, static and dynamic. Sprites drawn during the static pass do not change between frames; sprites drawn during the dynamic pass change every frame. The static pass is rendered

into the refresh buffers, and the dynamic pass is rendered into the composition buffers. The refresh buffers are used to "clear" the composition buffers.

Currently all objects and polygons are drawn during the dynamic pass, and terrain, walls, and floors are rendered during the static pass.

TBD: When an object changes its graphics in any way, it automatically becomes dynamic if it was static. If an object is dynamic and its graphics don't change for *n* frames (where *n* is tunable), it automatically reverts back to static.

10.2.6 Optimizations

The primary 2D optimization to be made is much faster z-sprite blitting. TBD: This may be accomplished through:

- hand-optimized blitters
- caching common sprites in an uncompressed form
- reading the z-buffer only when necessary

10.2.7 Hardware Acceleration

In the absence of z-blitters, it is unlikely we will benefit from 2D hardware acceleration.

10.3 3D Graphics

10.3.1 DDD

DDD is a C++ library which thinly wraps Direct3D.

TBD: Will we continue to use DDD, or will we port to 3RASH or J3, or will be port the bottom-end of DDD to one of these solutions?

TBD: DDD may require some work to provide the desired frame rate. In particular, bundling together multiple meshes into a single execute buffer, or abandoning execute buffers altogether, may be required. Another possibility is to sort meshes by texture/material. This could be done inside or outside DDD.

TBD: DDD uses Direct3D's transform and lighting modules. We may be have to modify DDD to allow us to take over the transform and/or lighting modules.

10.3.2 View Transform

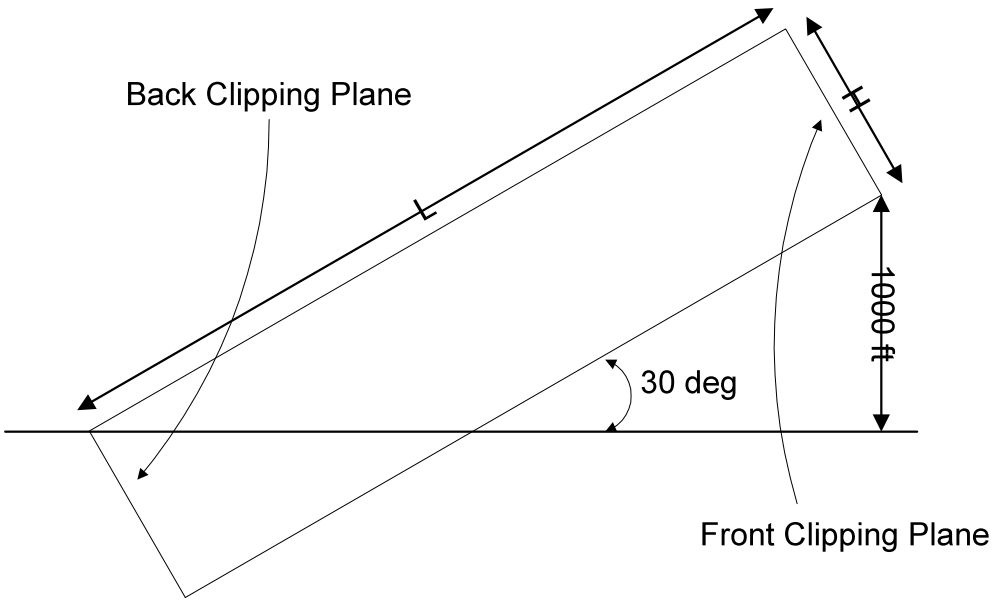
The view transform is constructed to place the camera at a 30° angle above the horizon. The camera is positioned to point at the center pixel of the center-most tile (assumed to be at 0 altitude). When the user "rotates the world," the view transform is modified to reflect a new camera position.

10.3.3 Projection Transform

The projection transform performs an orthographic projection from view space to a "canonical view volume." The canonical view volume maps to a view volume oriented in the world (see below figure). The view volume is constructed so that the back clipping plane never clips anything at 0 altitude. The bottom of the front clipping plane is located 1000 feet above 0 altitude, so that nothing gets clipped in front unless it is more than 1000 feet above 0 altitude.

10.3.3.1 Z-Buffer Resolution

Since the projection is non-perspective, the z-buffer is linear. For a z-buffer with 16 bits per pixel, z-buffer "units" are approximately 1cm.



View Volume

10.3.4 Models

Each polygonal mesh is defined in terms of its own model coordinates. DDD's hierarchical transform stack allows composing meshes into groups of meshes by concatening model-to-world transforms.

10.3.5 Texture Maps

Each mesh may have a single texture applied to it. Allowed texture sizes are 64x64, 128x128, or 256x256 pixel. Texture maps are 8-bit palettized BMPs.

TBD: DDD may have to be modified to allow changing texture coordinates on-the-fly.

10.3.6 Mesh Deformations

DDD does not support mesh deformations.

TBD: Do we need them?

10.3.7 Optimizations

Possible optimizations include:

- Packing more than a single mesh into an execute buffer
- Sorting meshes by texture
- Using a custom software renderer (since we don't need anything fancy, like perspective-correct texture mapping).
- Performing the transform and lighting stages of the pipeline ourselves.

10.3.8 Hardware Acceleration

DDD supports hardware acclereation, but Jefferson does not yet. Hardware acceleration on a good 3D card will enable:

- Opportunities for parallelism
- Translucent polygons
- Drawing terrain as a polygonal mesh, possibly every frame

- Drawing walls as polygons
- Smooth rotations, during which z-buffer sprites are rendered as sprite decals.

Hardware acceleration presents some risks, as well. Among them:

- PCI bus bandwidth limitations. Due largely to the fact that there is no z-buffer blitting in DirectX.

10.3.9 Terrain

TBD: How terrain will be drawn.

10.4. Lighting

10.4.1 Polygons

Polygons will be lit using D3D lighting module. If this proves inadequate, we will have to implement our own.

TBD: Assuming the D3D lighting model is used, the final lighting solution for polygons will probably consist of a combination of ambient and diffuse lighting effects. Specular highlighting and ambient light are probably not useful enough to use, and their performance from 3D accelerator to 3D accelerator varies signficantly.

TBD: A character's "ambient" light may be achieved using D3D's ambient light, or it may require moving D3D light objects around the person

10.4.2 Sprites

Sprites will be rendered lit by "standard" overhead light (as determined by the artist). The sprites will have virtually no directional lighting so they look good when rotated.

10.4.2.1 Sprite Ambient Lighting

In the absence of directional lighting (see next section), sprites will be lit according to the ambient light of their surroundings.

TBD: As well as a "standard" palette, sprites will include a "darkest" palette. The "darkest" palette will be used to render the sprite when the ambient light is at its lowest value. For ambient light values between standard and darkest, we will interpolate between entries in the two palettes.

10.4.2.2 Sprite Directional Lighting

Sprites may be directionally lit by lamps or other light sources.

The first approach will be to generate a "normal map" at sprite generation time which encodes the normal at each pixel. When that pixel is to be drawn, it's normal information is used to determine how bright or dark the pixel will look to the viewer.

At runtime, only the light sources "near" an object will contribute to its directional lighting. Their distance and position relative to the sprite will be used to brighten or darken each pixel based on which direction that pixel is facing.

Note: it may require MMX to do this quickly enough.

Risks:

- May results in weird "light bands" across the sprite.

TBD: We may also need some encoded material information.

10.4.3 Day/Night

Day and night will be indicated by changing the ambient light, both for sprites and polygons.

10.4.4 Shadows

TBD: Polygons may have shadows.

TBD: Sprites will not have shadows.

10.4.5 Glows

There are two kinds of glow to consider, radiant and emissive. Radiant glows light other things, such as when a refrigerator is opened or TV is on in a dark room. Emissive glows, such as the glow of a stereo's LED, does not light other things.

Radiant glows will be done on an object-by-object basis using transparent polygons. In software, this will most likely be done with stippling; on 3D hardware this will be done using alpha blending.

Emissive glows will be done using palette tricks. This will require a mechanism for *not* darkening specific palette entries when generating night time palettes from standard palettes. TBD: How best to accomplish this.

10.4.6 Walls

TBD: How to light walls in software

TBD: In hardware, walls will be lit by making a second rendering pass modulating a light map on to the wall pixels.

10.5 Picking

10.5.1 Sprite Picking

TBD: The required picking resolution.

Sprites are picked by mapping a mouse click and the z-buffer value under it to a point in world coordinates. The pick resolution is approximately 8 mm in world coordinates.

10.5.1.1 Per-Object Picking

If we need to get more complicated, we can do per-object picking by blitting each component sprite in turn, and examining the z-value at the point in question.

TBD: How complicated do we need to get?

10.5.2 Polygon Picking

Direct3D supports polygon picking. DDD will be modified to use the Direct3D implementation if polygon picking is needed.

TBD: Is polygon picking needed?

10.6 Highlighting

10.6.1 Objects

Sprites are highlighted by blitting them using a "brighter" palette. The so-called "sprite-brite" palette is generated from the default palette by converting each RGB value to the YUV color space, increasing its brightness, and converting back to RGB.

10.6.2 People

Currently people are lit by "flashing" the lights on them.

TBD: How to highlight people?

One possibility is to use textures that are extremely bright, and render people in low-light conditions most of the time. The selected person would then be rendered extremely brightly. Unfortunately, this leads to huge ramps when using the ramp HEL Direct3D device.