

Rendering With Coherent Layers

Jed Lengyel and John Snyder
Microsoft Research

Abstract

For decades, animated cartoons and movie special effects have factored the rendering of a scene into layers that are updated independently and composed in the final display. We apply layer factorization to real-time computer graphics. The layers allow targeting of resources, whether the ink and paint artists of cartoons or the graphics pipeline as described here, to those parts of the scene that are most important.

To take advantage of frame-to-frame coherence, we generalize layer factorization to apply to both dynamic geometric objects and terms of the shading model, introduce new ways to trade off fidelity for resource use in individual layers, and show how to compute warps that reuse renderings for multiple frames. We describe quantities, called *fiducials*, that measure the fidelity of approximations to the original image. Layer update rates, spatial resolution, and other quality parameters are determined by geometric, photometric, visibility, and sampling fiducials weighted by the content author’s preferences. We also compare the fidelity of various types of reuse warps and demonstrate the suitability of the affine warp.

Using Talisman, a hardware architecture with an efficient layer primitive, the work presented here dramatically improves the geometric complexity and shading quality of scenes rendered in real-time.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Additional Keywords: sprite, affine transformation, image compositing, image-based rendering, Talisman

1 Introduction

The layered pipeline separates or *factors* the scene into layers that represent the appearance of an object (e.g., a space ship separate from the star field background) or a special lighting effect (e.g., a shadow, reflection, highlight, explosion, or lens flare.) Each layer produces a 2D-image stream as well as a stream of 2D transformations that place the image on the display. We use *sprite* to refer to a layer’s image (with alpha channel) and transformation together.

The layered pipeline decouples rendering of layers from their display. Specifically, the sprite transformation may be updated more frequently than the sprite image. Rendering (using 3D CG) updates the sprite image only when needed. Sprite transforming and compositing [Porter84] occur at display rates. The sprite transformation scales low-resolution sprites up to the display resolution, and transforms sprites rendered earlier to approximate their later appearance. In other words, the sprite transformation interpolates rendered image streams to display resolution in both space and time.

Layered rendering has several advantages for real-time CG. First, layered rendering better exploits coherence by separating fast-moving foreground objects from slowly changing background layers. Second, layered rendering more optimally targets rendering resources by allowing less important layers to be degraded to conserve resources for more important layers. Finally, layered rendering naturally integrates 2D elements such as overlaid video, offline rendered sprites, or hand-animated characters into 3D scenes.

As an architectural feature, decoupling rendering from compositing is advantageous. Compositing is 2D rather than 3D, requires no

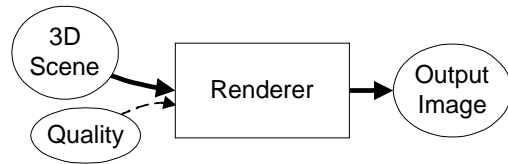


Figure 1: TRADITIONAL PIPELINE processes the entire scene database to produce each output image. The quality parameters for texture and geometry (such as level-of-detail) may be set independently for each object in the scene. However, the sampling resolutions in time (frame rate) and space (image resolution and compression) are the same for all objects in the scene.

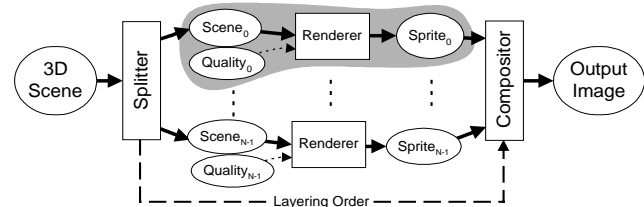


Figure 2: LAYERED PIPELINE partitions the scene into independent layers. A single layer’s pipeline (highlighted at top) is similar to the traditional pipeline. By adjusting each layer’s quality controls, the content author targets rendering resources to perceptually important parts of the scene. Slowly changing or unimportant layers are updated at lower frame rates, at lower resolution, and with higher compression.

z-buffer, no lighting computations, no polygon edge antialiasing, and must handle few sprites (which are analogous to texture-mapped polygons) relative to the number of polygons in the rendered geometry. This simplicity allows compositing hardware to be made with pixel fill rates much higher than 3D rendering hardware. Our investigation demonstrates that the saving in 3D rendering justifies the extra hardware expense of a compositor.

The layered pipeline augments the set of traditional rendering quality parameters such as geometric level-of-detail and shading model (e.g., flat-, Gouraud-, or Phong-shaded), with the temporal and spatial resolution parameters of each layer. The *regulator* adjusts the quality parameters in order to achieve optimal quality within fixed rendering resources. The regulator dynamically measures both the costs of changing the quality parameters – how much more or less of the rendering budget they will consume – and the benefits – how much improvement or loss in fidelity will occur.

The specific contributions of this paper include extending the generality of factoring. While some authors have considered factoring over static geometric objects [Regan94, Maciel95, Shade96, Schaufler96ab], we consider dynamic situations and factoring over shading expressions (Section 2). We describe how to render using the layered pipeline (Section 3). We investigate different types of sprite transformations and show why an affine transformation is a good choice (Section 4). We discuss low-computation measures of image fidelity, which we call *fiducials*, and identify several classes of fiducials (Section 5). We add the spatial and temporal resolution of layers as regulation parameters and propose a simple regulator that balances them to optimize image fidelity (Section 6). Finally, we demonstrate that the ideas presented here enhance performance of the Talisman architecture by factors of 3-10, by using interpolated triple-framing or by regulating the heterogeneous update of sprite images (Section 7).

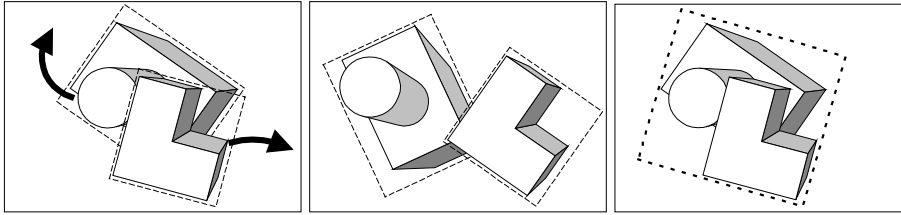


Figure 3: INDEPENDENT UPDATE depends on choice of factoring. The left and middle figures show how factoring the geometry into two separate layers allows each layer to be reused. The right figure shows a less effective partition.

1.1 The Layered Pipeline and Talisman

The Talisman reference hardware architecture was designed to support the layered pipeline (see [Torborg96] for details.) Rendering occurs within one 32×32 chunk at a time, so that z-buffer and fragment information for antialiasing can be stored on-chip. The resulting sprites with alpha channel are compressed and written to sprite memory. In parallel with 3D rendering, for every frame, the compositor applies an affine warp to each of an ordered set of sprites uncompressed from sprite memory and composites the sprites just ahead of the video refresh, eliminating the need for a frame buffer. Sprite composition is limited to the “over” operator [Porter84].

Although our experiments assume the Talisman reference architecture, the ideas can be usefully applied to traditional architectures. Layer composition can be emulated with rendering hardware that supports texture mapping with transparency by dedicating some of the pixel-fill rate of the renderer for sprite composition. This may be a good sacrifice if sprite rendering is polygon limited rather than pixel fill limited. Clearly though, Talisman is a superior layered pipeline in that sprite composition is “for free” (i.e., sacrifices few rendering resources) and very high speed (because of sprite compression and the simplicity of sprite composition in relation to rendering).¹

1.2 Previous Work

To avoid visibility sorting of layers, alternative architectures use what are essentially sprites with z information per pixel [Molnar92, Regan94, Mark97]. Such systems are more costly in computation, bandwidth, and storage requirements since z must be stored and transmitted to a more complicated compositor. Z information is also difficult to interpolate and compress. We observe that z information per pixel is greatly redundant when used solely to determine a layering order. But such an ordering is necessary to ensure an antialiased result.² Our approach of factoring into layers allows warping per coherent object. It also avoids problems with uncovering of depth-shadowed information. Of course, sprites could store multiple z layers per pixel, a prohibitively costly approach for hardware, but one near to ours in spirit. Such a scheme stores all the layers within each pixel, rather than all the pixels for each layer.³

[Funkhouser93] adjusts rendering parameters to extract the best quality. We add sprite resolution and update rate to the set of regulated parameters and make photometric measurements rather than relying on *a priori* assignment of benefit to sampling rate. [Maciel95] takes a similar approach but use fiducials and impostor representations optimized for walkthroughs of static scenes.

Taking advantage of temporal coherence has been an ongoing theme in computer graphics [Hubschman81, Shelley82]. [Hofmann89] presents techniques for measuring how much camera movement is

¹ In a prototype implementation of Talisman, the compositor is planned to run at 320M pixels/second compared to 40Mps for the renderer.

² Penetrating z-sprites will have a point-sampled and thus aliased boundary where visibility switches.

³ Post-warping of unfractured z-images also fails to address the case of independently moving objects.

allowed before changes in the projected geometry exceed a given tolerance. [Chen93, Chen95] show how to take advantage of coherence between viewpoints to produce nearly constant cost per frame walkthroughs of static environments. [McMillan95] re-projects images to produce an arbitrary view.

Our use of temporal coherence is most similar to [Regan94], who observed that not all objects need updating at the display rate. Rather than factoring globally across

object sets requiring a common update rate, our scheme factors over geometric objects and shading model terms, and accounts for relative motion between dynamic objects.

[Shade96] and [Schaufler96ab] use image caches and texture-mapped quadrilaterals to warp the image. This is conceptually similar to our work, but does not include the factoring across shading or the idea of real-time regulation of quality parameters. We harness simpler image transformations (affine rather than perspective) to achieve greater fidelity (Section 4.2). Our work also treats dynamic geometry.

Shading expressions [Cook84, Hanrahan90] have been studied extensively. [Dorsey95] factors shading expressions by light source and linearly combines the resulting images in the final display. [Guenther95] caches intermediate results. [Meier96] uses image processing techniques to factor shadow and highlight regions into separate layers which are then re-rendered using painterly techniques and finally composited. The novel aspects of our technique are the independent quality parameters for each layer and warp of cached terms.

2 Factoring

The guiding principle of our approach is to factor into separate layers elements that require different spatial or temporal sampling rates. This section discusses guidelines for manually factoring across geometry and shading, visibility sorting of layers, and annotating models with layer information.

2.1 Factoring Geometry

Geometry factoring should consider the following properties of objects and their motions:

1. *Relative velocity* – A sprite that contains two objects moving away from each other must be updated more frequently than two sprites each containing a single object (Figure 3). Relative velocity also applies to shading.
2. *Perceptual distinctness* – Background elements require fewer samples in space and time than foreground elements, and so must be separated into layers to allow independent control of the quality parameters.
3. *Ratio of clear to “touched” pixels* – Aggregating many objects into a single layer typically wastes sprite area where no geometry projects. Finer decompositions are often tighter. Reducing wasted sprite space saves rendering resources especially in a chunked architecture where some chunks can be eliminated, and makes better use of the compositor, whose maximum speed limits the average depth complexity of sprites over the display.

2.2 Visibility Sorting

Visibility sorting of dynamic layer geometry can be automated. We have implemented a preliminary algorithm for which we provide a sketch here. A full discussion along with experimental results is in progress [Snyder97].

To determine visibility order for layers containing moving geometry, we construct an incrementally changing kd-tree based on a set of constant directions. A convex polyhedron bounds each layer’s geometry, for which we can incrementally compute bounding extents

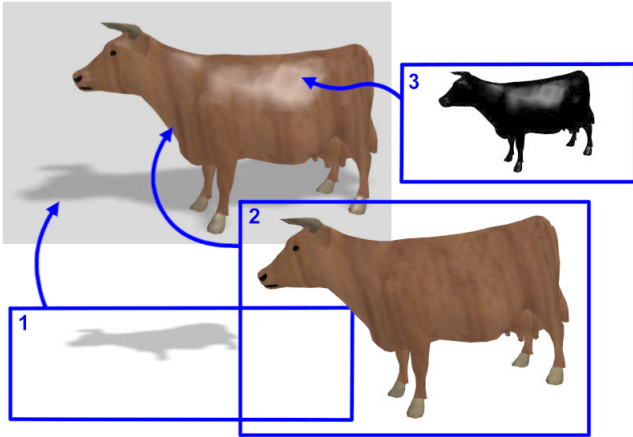


Figure 4: **FACTORED SHADING EXPRESSION** separates shadow, diffuse, and specular terms. In this example, the shadow and specular sprites are both computed at 25% (50% in x and y) of the display resolution. The shadow sprite modulates the color. The specular sprite adds to the output without changing alpha.

in each direction. The kd-tree quickly determines the set of objects that can possibly occlude a given object, based on these extents. Using this query, the visibility sort computes an incremental topological sort on the strongly connected components (which represent occlusion cycles) of the occlusion graph. Strongly connected components must be temporarily aggregated in the same layer, since a priority ordering does not exist.⁴

2.3 Factoring Shading

Shading may also be factored into separate layers. Figure 4 shows a typical multipass example, in which a shadow layer modulates the fully illuminated scene, and a reflection layer adds a reflection (in this case the reflection is the specular reflection from a light.) Figure 5 shows a schematic view of the steps needed to create the multipass image. The shadow layer is generated from a depth map rendered from the point of view of a light. The reflection layer is generated from a texture map produced by a separate rendering with a reflected camera. The layers shown in the figure represent post-modulation images using the same camera. With traditional architectures, the three layers are combined in the frame buffer using pixel blend operations supported by the 3D hardware, as described in [Segal92].

Shadows and reflections may instead be separated into layers as shown in Figure 6, so that the blend takes place in the compositor rather than the renderer. We call these *shade sprites* in reference to shade trees [Cook84]. To take advantage of temporal coherence, highlights from fast moving lights, reflections of fast moving reflected geometry, and animated texture maps should be in separate layers and rendered at higher frame rates than the receiving geometry. To take advantage of spatial coherence, blurry highlights, reflections, or shadows should be in separate layers and given fewer pixel samples.

For reflections, the correctness of using the compositor is evident because the reflection term is simply added to the rest of the shading. More generally, any terms of the shading expression that are combined with '+' or 'over' may be split into separate layers. 'A + B' can be computed using 'A over B' and setting A's alpha channel to zero.

The separation of shadows is slightly more difficult. The shadowing term multiplies each part of the shading expression that depends on a given light source. Many such terms can be added for

⁴ At least, an ordering does not exist with respect to hulls formed by the set of bounding directions, which is a more conservative test than with the original bounding polyhedra. Note that visibility order for aggregated layers is computed simply by rendering into the same hardware z-buffer.

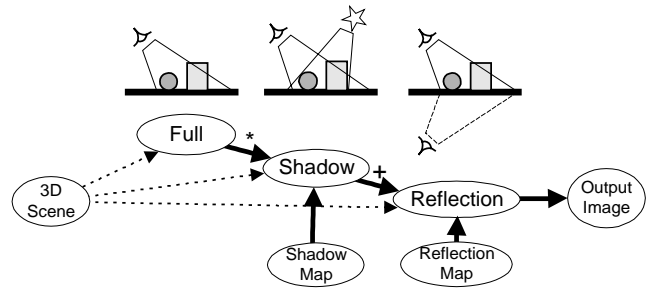


Figure 5: **MULTIPASS RENDERING** combines the results of several rendering passes to produce effects such as shadows and reflections. With a traditional architecture, the rendering passes are combined using blending operations in the 3D renderer (multiplication for shadow modulation and addition for adding reflections.)

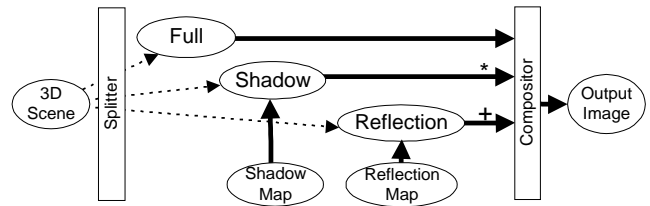


Figure 6: **SHADE SPRITES** are combined in the final composition phase to produce the multipass rendering. Each shading term may have different resolutions in space and time.

multiple shadowing light sources. We describe an approximation to multiplicative blending using the 'over' composition operator in an appendix.

Consider a simple example of a shading model with two textures and a shadow, $S(N \cdot L)(T_1 + T_2)$, where S is the shadowing term, N is the normal to the light, L is the light direction, and T_1 and T_2 are texture lookups. This shading model can be factored into three layers: S , $(N \cdot L)T_1$, and $(N \cdot L)T_2$, which are composited to produce the final image. The fact that this expression can be reordered and partial results cached is well known [Guenther95]. What we observe here is that each of these factors may be given different sampling resolutions in space and time, and interpolated to display resolutions.

As an aside, we believe shade sprites will be useful in authoring. When modifying the geometry and animation of a single primitive, the artist would like to see the current object in the context of the fully rendered and animated scene. By pre-rendering the layers that are not currently being manipulated, the bulk of the rendering resources may be applied to the current layer. The layers in front of the current layer may be made partially transparent (using a per-sprite alpha multiplier) to allow better manipulation in occluded environments. By using separate layers for each texture shading term, the artist can manipulate the texture-blending factors interactively at the full frame rate.

2.4 Model Annotation

The first step of model annotation is to break the scene into "parts" such as the base level joints in a hierarchical animated figure. The parts are containers for all of the standard CG elements such as polygon meshes, textures, materials, etc., required to render an image of the part. A part is the smallest renderable unit.

The second step is to group the parts into layers according to the guidelines described above. The distinction is made between parts and layers to allow for reuse of the parts, for example in both a shadow map layer and a shadow receiver layer.

The final step is to tag the layers with resource-use preferences relative to other layers in the scene. The preferences are relative so that total resource consumption can change when, for example, other applications are started (as discussed in Section 6).

3 Image Rendering

This section discusses how a layer's sprite image is created (i.e., rendered). Once created, the image can be warped in subsequent frames to approximate its underlying motion, until the approximation error grows too large. Although the discussion refers to the Talisman reference architecture with its 2D affine image warp, the ideas work for other warps as well.

3.1 Characteristic Bounding Polyhedron

The motion of the original geometry is tracked using a *characteristic bounding polyhedron*, usually containing a small number of vertices (Figure 7). For rigidly moving objects, the vertices of the characteristic polyhedron, called *characteristic points*, are transformed using the original geometry's time-varying transform. Nonrigidly deforming geometry can be tracked similarly by defining trajectories for each of the characteristic points. To group rigid bodies, we combine the characteristic bounding polyhedra, or calculate a single bounding polyhedron for the whole.

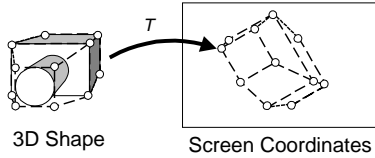


Figure 7: CHARACTERISTIC BOUNDING POLYHEDRON matches the shape of the geometry but has fewer vertices.

be chosen by embedding preferred axes in the original model, and transforming the axes to screen space.

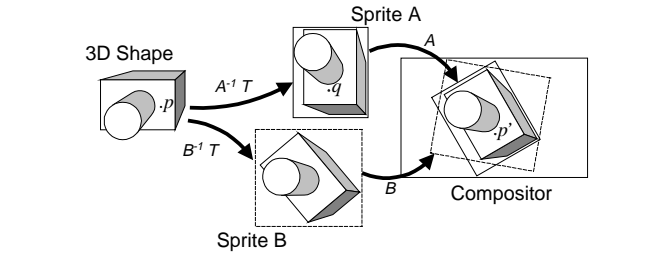


Figure 9: SPRITE RENDERING TRANSFORMATION maps the 3D shape into the sprite image. Affine transform B does not make the best use of image samples, while A fits the projected shape tightly.

Using the bounding slabs, we find the bounding rectangle with the smallest area (Figure 10). The origin and edges of the rectangle determine the affine matrix. Initially, we searched for the smallest area parallelogram, but found the resulting affine transformation had too much anisotropy.

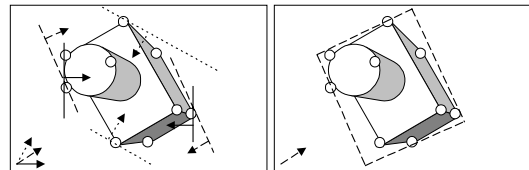


Figure 10: BOUNDING SLABS are obtained by taking the extremal values of the dot product of each slab direction with the characteristic points. A tight-fitting initial affine transform can be calculated by taking the minimum area rectangle or parallelogram that uses the slab directions.

3.2 Sprite Extents

For a particular frame, there is no reason to render off-screen parts of the image. But in order to increase sprite reuse, it is often advantageous to expand the sprite image to include some off-screen area.

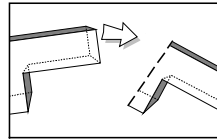


Figure 8a shows how clipping a sprite to the screen (solid box) prevents its later reuse because parts of the clipped image later become visible. In Figure 8b, the sprite extent (dashed box) has been enlarged to include regions that later become visible. The extra area to include depends on such factors as the screen velocity of the sprite (which suggests both where and how much the extents should be enlarged) and its expected duration of reuse.

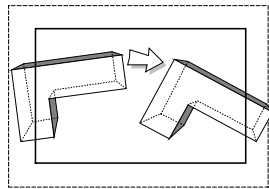


Figure 8: SPRITE EXTENTS enlarge the display extent to reuse sprites whose geometry lies partially off-screen.

3.4 Spatial Resolution

The choice of affine matrix A also determines how much the sprite is magnified on the display. Rendering using a sampling density less than the display's is useful for less important objects or for intentional blurring (Figure 11). The default is to use the same sampling density as the screen, by using the length in pixels of each side of the parallelogram from Section 3.3. See Figure 24 for an example of different sampling resolutions per sprite.

For a linear motion blur effect, the sprite sampling along one of the axes may be reduced to blur along that axis. The sprite rendering transformation should align one of the coordinate axes to the object's velocity vector by setting the bounding slab directions to the velocity vector and its perpendicular.

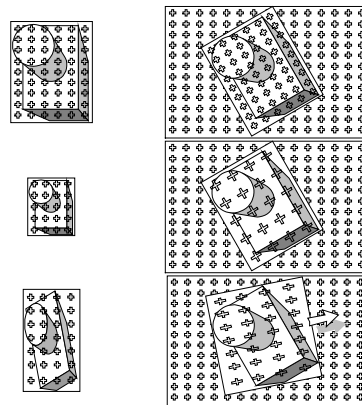


Figure 11: SPATIAL RESOLUTION is independent of the display resolution. The sampling density of the top sprite is the same as the screen. The middle sprite uses fewer samples than the screen, trading off pixel fill for blur. The bottom sprite aligns to the velocity vector and uses fewer samples along one dimension for a motion blur effect.

3.3 Sprite Rendering Transformation

When creating a sprite image, we must consider a new transform in the pipeline in addition to the modeling, viewing, and projection transforms: a 2D affine transform that maps the sprite to the screen.

If T is the concatenation of the modeling, viewing, and projection matrices, a screen point p' is obtained from a modeling point p , by $p' = Tp$. For the sprite transformation, $p' = Aq$ where A is an affine transform and q is a point in sprite coordinates. To get the proper mapping of geometry to the display, the inverse 2D affine transform is appended to the projection matrix, so that $q = A^{-1}Tp$ results in the same screen point $p' = Aq = AA^{-1}Tp = Tp$ (Figure 9). The choice of matrix A determines how tightly the sprite fits the projected object. A tighter fit wastes fewer samples as discussed in Section 2.

To choose the affine transform that gives the tightest fit, we first project the vertices of the characteristic bounding polyhedron to the screen, clipping to the expanded sprite extent. Then, using discrete directions (from 2-30, depending on the desired tightness), we calculate 2D bounding slabs [Kay86]. Alternately, the slab directions may

4 Image Warps

To reuse a rendered sprite image in subsequent frames, an image warp is used to approximate the actual motion of the object. We use the projected vertices of the bounding polyhedron (the characteristic points) to track the object's motion, as shown in Figure 12.

To reuse images where objects are in transition from off-screen to on-screen, and to prevent large distortions (i.e., ill-conditioning of the resulting systems of equations), the characteristic bounding polyhedron is clipped to the viewing frustum, which may be enlarged from the display's as discussed in Section 3.2. The clipped points are added to the set of characteristic points (Figure 13) and used to determine an approximating sprite transformation as described below.

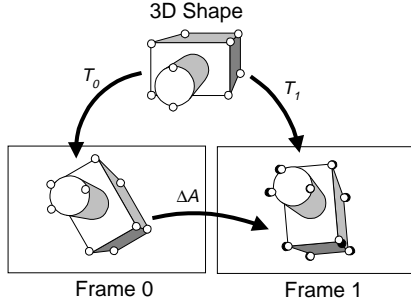


Figure 12: MATCHING CHARACTERISTIC POINTS ON THE 3D shape are projected to the screen to find a transform A that best matches the original points (white) to the points in the new frame (black).

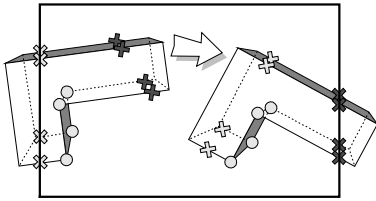


Figure 13: CLIPPED CHARACTERISTIC POLYHEDRON adds corresponding points introduced by clipping the characteristic polyhedron at the last-rendered and current frames.

hedron vertices, ignoring the z values and adding a row of 1's to account for the translation

$$P = \begin{bmatrix} x_0 & \dots & x_{n-1} \\ y_0 & \dots & y_{n-1} \\ 1 & \dots & 1 \end{bmatrix}$$

where n is the number of points (at least 3 for the affine transform).

Let \hat{P} be the matrix of characteristic points at the initial time and P be the matrix at the desired time t . We solve for the best least-squares transform that matches the two sets of image-space points [Xie95].

In an affine transform, the x and y dimensions are decoupled and so may be solved independently. To solve $A\hat{P} = P$ at time t for the best A , in the least-squares sense, we use normal equations:

$$A\hat{P}\hat{P}^T = P\hat{P}^T \\ A = P\hat{P}^T(\hat{P}\hat{P}^T)^{-1}$$

The normal-equations technique works well in practice, as long as the projected points are reasonably distributed. Adding the clipped characteristic points ensures that $\hat{P}\hat{P}^T$ is not rank deficient. Much of the right hand side may be collected into a single vector K that may be reused for subsequent frames.

$$K = \hat{P}^T(P\hat{P}^T)^{-1} \\ A = PK$$

To calculate K requires the accumulation and inverse of a symmetric 3×3 matrix.

4.1 Affine Warp

A 2D affine transform is represented by a 2×3 matrix, where the right column is translation and the left 2×2 is the rotation, scale, and skew.

$$A = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix}$$

Let P be the time-varying set of projected and clipped bounding poly-

4.2 Comparison of Warps

Clearly, other types of image warps can be used in place of the affine described above. In order to compare alternative image warps, we ran a series of experiments to

1. measure update rate as a function of maximum geometric error for various warps, and
2. measure perceptual quality as a function of update rate for various warps.

Each series involved the animation of a moving rigid body and/or moving camera to see how well image warping approximates 3D motion. We tried several types of rigid bodies, including nearly planar and non-planar examples. We also tried many animated trajectories for each body including translations with fixed camera, translations accompanied by rotation of the body along various axes with various rotation rates, and head turning animations with fixed objects.

The types of 2D image warps considered were

1. pure translation,
2. translation with isotropic scale,
3. translation with independent scale in x and y ,
4. general affine, and
5. general perspective.

The fundamental simulation routine computes an animation given a geometric error threshold, attempting to minimize the number of renderings by approximating with an image warp of a particular type. A pseudo-code version is shown in Figure 14.

```
simulate(error-threshold, warp-type, animation)
{
  for each frame in animation
    compute screen position of characteristic points at current time
    compute transform (of warp-type) which best maps old
      cached positions to new positions
    compute maximum error for any characteristic point
    if error exceeds threshold
      re-render and cache current positions of characteristic points
    else
      display sprite with computed transformation
    endif
  endfor
  return total number of re-renderings
}
```

Figure 14: EXPERIMENT PSEUDOCODE shows steps used to compute update rates of various warps.

Ideally, we would like to compute approximations of each type that minimize the maximum error over all characteristic points, since this is the regulation metric. This is a difficult problem computationally, especially since the warping transformation happens at display rates for every layer in the animation. Minimizing the sum of squares of the error is much more tractable, yielding a simple linear system as we have already discussed. As a compromise, we simulated both kinds of error minimization: sum-of-squares and maximum error using an optimization method for L^∞ norms that iteratively applies the sum-of-square minimization, as described in [Gill81, pp. 96-98].

Further complicating matters, minimizing the error for perspective transformations is easier when done in homogeneous space rather than 2D space, again since the latter yields an 8×8 linear system rather than a difficult nonlinear optimization problem. We therefore included sum-of-square and maximum error methods for the first four (non-perspective) transformation types.

For perspective, we included sum-of-square minimization in homogeneous space (yielding a linear system as described above), maximum error in homogeneous space (using the technique of [Gill81]), and sum-of-square minimization in nonhomogeneous space (post-perspective divide), using gradient descent.⁵ The starting point

⁵ Minimization of the maximum nonhomogeneous error seemed wholly impractical for real-time implementation.

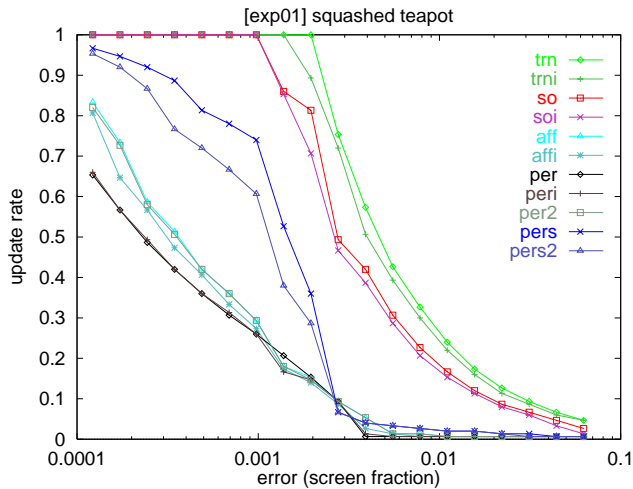


Figure 15: FLAT TEAPOT update-rate/error relations for the various warps show a surprisingly small difference between the affine and the perspective for a nearly flat object .

for the gradient descent was the sum-of-squares-error-minimizing affine transformation.

We also included a perspective transformation derived using the method of [Shade96], in which objects are replaced by a quadrilateral placed perpendicular to the view direction and through the center of the object’s bounding box. Our derivation projects the characteristic points onto this quadrilateral, bounds the projected points with a rectangle, and projects the corners of the rectangle to the screen. The perspective transformation that maps the old corners of the rectangle to their current locations is selected as the approximation. In yet another version, Shade’s method is used as a starting point and then refined using gradient descent in nonhomogeneous space with the sum-of-square error metric.

Representative results of the first series of experiments are shown in Figure 15 and Figure 16. In both figures, the experiment involved a rotating and translating teapot which is scaled nearly flat⁶ in Figure 15 and unscaled in Figure 16. Error thresholds ranging from 1/8 pixel to 64 pixels were used for each warp type/error minimization method, assuming an image size of 1024×1024, and the resulting rate of re-rendering measured via the simulation process described above. The meanings of the curve name keywords are as follows:

keyword	Warp type and minimization method
trn	translation, sum-of-square
trni	translation, max
so	translation with xy scale, sum-of-square
soi	translation with xy scale, max
aff	affine, sum-of-square
affi	affine, max
per	perspective, homogeneous, sum-of-square
peri	perspective, homogeneous, max
per2	perspective, nonhomogeneous, sum-of-square
pers	perspective, method of Shade
pers2	pers, followed by gradient descent

In the case of the flat teapot (Figure 15), note that the error/update curves cluster into groups – translation, translation with separate scale, Shade, affine, and perspective, in order of merit. Shade’s method is significantly outperformed by affine. Note also that the

⁶ The teapot, a roughly spherical object, was scaled along its axis of bilateral symmetry to 5% of its previous size.

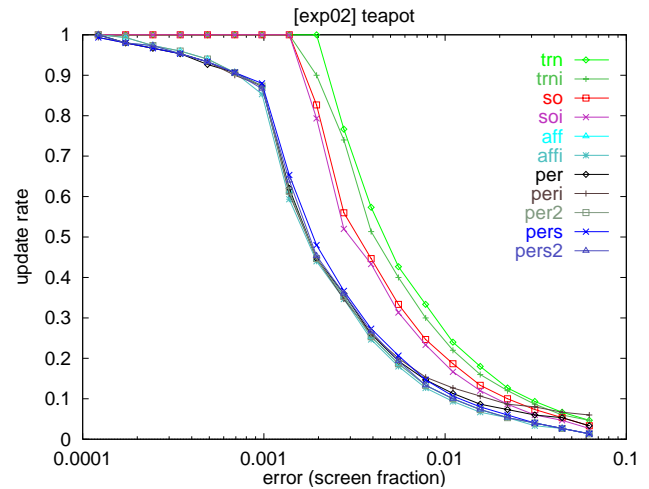


Figure 16: REGULAR TEAPOT update-rate/error relations show that affine and perspective are nearly indistinguishable.

sum-of-square error minimization is not much different than maximum error minimization for any of the warp types. The difference between perspective and affine is much less than one might expect in this case, given that perspective exactly matches motions of a perfectly flat object. Figure 16 (regular teapot) is similar, except that the clusters are translation, translation with separate scale, and all other warp types. In this case, perspective yields virtually no advantage over affine, and in fact is slightly worse towards the high-error/low update rate end of the curves for the homogeneous space metrics (per and peri).⁷ This is because the homogeneous metric weights the errors unevenly over the set of characteristic points. The method of Shade is slightly worse than affine in this case.

Since geometric error is a rather indirect measure of the perceptual quality of the warp types, the second series of experiments attempted to compare the perceptual quality of the set of warps given an update rate (i.e., an equal consumption of rendering resources). We used binary search to invert the relation between error threshold and update rate for each warp type, and then recorded the same animation, at the same update rate⁸, for various image warp approximations. Although subjective, the results confirm the merit of the affine transformation over less general transformations and the lack of improvement with the more general perspective transformation in typical scenarios.

4.3 Color Warp

Images can be “warped” to match photometry changes as well as geometry changes. For example, Talisman provides a per-sprite color multiplier that can be used to match photometry changes. To solve for this multiplier, we augment each characteristic point with a normal so that shading results can be computed (see Section 5.2). The color multiplier is selected using a simple least-squares technique that best matches the original color values of the shaded characteristic points to the new color values.

⁷ In the second series of experiments, the animations that used the homogeneous-weighted metric to determine an approximating perspective transformation looked visibly worse than those that used the simple affine transformation.

⁸ The update rate is the fraction of frames re-rendered; this balances the total consumption of rendering resources over the whole animation.

5 Fiducials

Fiducials measure the fidelity of the approximation techniques. Our fiducials are of four types. Geometric fiducials measure error in the screen-projected positions of the geometry. Photometric fiducials measure error in lighting and shading. Sampling fiducials measure the degree of distortion of the image samples. Visibility fiducials measure potential visibility artifacts.

We use conservative measurements where possible, but are willing to use heuristic measurements if efficient and effective. Any computation expended on warping or measuring approximation quality can always be redirected to improve 3D renderings, so the cost of computing warps and fiducials must be kept small relative to the cost of rendering.

5.1 Geometric Fiducials

Let \hat{P} be a set of characteristic points from an initial rendering, let P be the set of points at the current time, and let W be the warp computed to best match \hat{P} to P . The geometric fiducial is defined as

$$F_{geom} = \max_i \|P_i - W\hat{P}_i\|$$

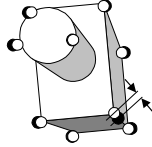


Figure 17: GEOMETRIC FIDUCIAL measures maximum pointwise distance between the warped original and current characteristic points.

5.2 Photometric Fiducials

We use two approaches to approximately measure photometric errors. The first uses characteristic points augmented with normals as described in Section 4.3 to point sample the lighting. Let \hat{C} be the colors that result from sampling the lighting at the characteristic points at the initial time, and C be the sampled colors at the current time. Let W_C be the color warp used to best match \hat{C} to C ⁹. Then the shading photometric fiducial is defined to be the maximum pointwise distance from the matched color to the current color.

$$F_{photo} = \max_i \|C_i - W_C \hat{C}_i\|$$

Another approach is to abandon color warping and simply measure the change in photometry from the initial time to the current. Many measures of photometric change can be devised. Ours measures the change in the apparent position of the light. Let \hat{L} be the position of the light at the initial time and L be its position at the current time (accounting for relative motion of the object and light). For light sources far away from the illuminated object, we can measure the angular change from \hat{L} to L with respect to the object, and the change in distance to a representative object “center”. For diffuse shading, the angular change essentially measures how much the object’s terminator moves around the object, and the change in distance measures the increase or decrease in brightness. Light sources close to the object are best handled with a simple Euclidean norm. For specular shading, changes in the eye point can also be measured.

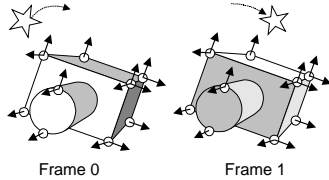


Figure 18: POINT-SAMPLED PHOTOMETRIC FIDUCIAL samples the shading at the initial and current characteristic points with normals.

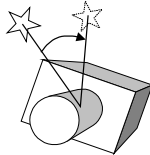


Figure 19: LIGHT SOURCE PHOTOMETRIC FIDUCIAL measures lighting change by the relative motion of light.

⁹ Note that in architectures without color warping capability, W_C is the identity transform and we simply measure the maximum shading difference over all the characteristic points.

5.3 Sampling Fiducials

Sampling fiducials measure distortion of the samples in the image approximation. In Figure 20, both the geometric and photometric fiducials indicate high fidelity, but the image is blurry. The magnitudes of the singular values of the Jacobian of the image mapping function measure the greatest magnification and minification and the ratio measures the maximum anisotropy¹⁰. The affine warp has a spatially invariant Jacobian given by the left 2x2 part of the 2x3 matrix, for

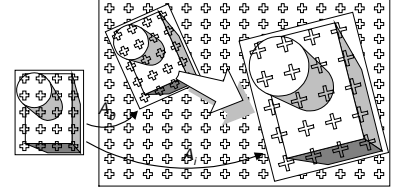


Figure 20: SAMPLING FIDUCIAL measures how the samples of a sprite are stretched or compressed.

which the two singular values are easily calculated [Blinn96]. For transforms with spatially varying Jacobians, such as the perspective warp, the singular values vary over the image. In this case, bounds on the singular values over the input domain can be computed.

5.4 Visibility Fiducials

Visibility fiducials measure potential visibility artifacts by tracking back-facing to front-facing transitions in the characteristic geometry (the simplified geometry makes these calculations tractable), and testing if the edges of clipped sprites become visible.

6 Regulation

A more complete treatment of regulation issues and directions may be found in [Horvitz96]. Our prototype regulator uses a simple cost-benefit scheduler and fiducial thresholds. The fiducial threshold provides a cutoff below which no attempt to re-render the layer is made (i.e., the image warp approximation is used). The regulator considers each frame separately, and performs the following steps:

1. Compute warp from previous rendering.
2. Use fiducials to estimate benefit of each warped layer.
3. Estimate rendering cost of each layer.
4. Sort layers according to benefit/cost.
5. Use fiducial thresholds to choose which layers to re-render.
6. Adjust parameters of chosen layers to fit within budget.
7. Render layers in order, stopping when all resources are used.

For a “budget-filling” regulator, the fiducial threshold is set to be small, on the order of a 1/1000 of the typical maximum error. All of the rendering resources are used in the attempt to make the scene as good as possible. For a “threshold” regulator, the threshold is raised to the maximum error that the user is willing to tolerate. This allows rendering resources to be used for other tasks.

Cost estimation [step 3] is based on a polygon budget, and measures the fraction of this budget consumed by the number of polygons in the layer’s geometry. Parameter adjustments [step 6] are made to the sprite’s spatial resolution using a budgeted total sprite size. This accounts for the rate at which the 3D rendering hardware can rasterize pixels.¹¹ Sprites that have been selected for re-rendering [step 5] are allocated part of this total budget in proportion to their desired area divided by the total desired area of the selected set. To dampen fluctuations in the regulation parameters which are perceptible when large, parameter changes are clamped to be no more than $\pm 10\%$ of their previous value at the time of last re-rendering. Note that factoring into many low-cost sprites allows smoother regulation.

¹⁰ The filtering capabilities of the hardware limit the amount of minification and anisotropy that can be handled before perceptible artifacts arise.

¹¹ A global average depth-complexity estimate is used to reduce the budget to account for rasterization of hidden geometry. Note that the depth complexity of factored geometry in a single layer is lower than would be the case for frame-buffer renderings of the entire scene.

7 Results

For the results presented here, we assume we have an engine that will composite all of the sprite layers with minimal impact on the rendering resources (as in the Talisman architecture.) We track the number of polygons and the amount of pixel fill used for rendering, but disregard compositing.

Both of the sequences described below are intended to represent typical content. For scenes with a moving camera, the typical speedup is 3-5 times what the standard graphics pipeline is capable of producing.

7.1 Canyon Flyby

This 250-frame sequence (Figure 25) used 10 sprite layers. We interpolated using affine warps and regulated the update rate with a geometric fiducial threshold of 4 pixels. Each sprite layer was re-rendered only when the geometric threshold was exceeded. The fiducial threshold of 4 pixels may seem large, but is acceptable for this sequence since the ships are well separated from the rest of the world.

The average update rate was 19%, and the cost-weighted average (based on polygon count) was 32%. About a third of the total polygons were rendered per frame.

In the canyon flyby scene, the entire background was placed in one sprite. Parallax effects from the rapidly moving camera make this a poor layering decision that yields a high update rate for the landscape sprite. In contrast, the sky is rendered just once

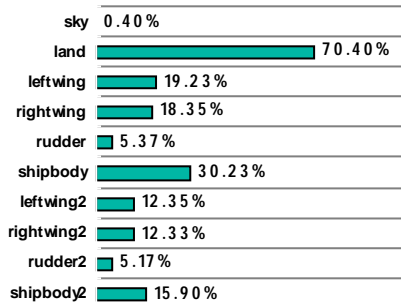


Figure 21: CANYON FLYBY AVERAGE UPDATE RATE for each sprite is the number of times each sprite was rendered divided by the number of frames in the sequence, 250

and then positioned with a new affine transformation each frame, and parts of the ships are updated relatively infrequently (5-30%).

7.2 Barnyard

The barnyard sequence (Figure 26) was chosen as an example in which a camera moves through a static scene¹². This is a difficult case, because the eye is sensitive to relative motion between static objects. Approximation errors in sequences in which objects already have relative motion are far less perceptible (e.g., the ships in the canyon flyby above.) Even with this difficult case, our interpolation technique is dramatically better than triple framing.

The scene is factored into 119 standard layers, 2 shadow map layers, and 2 shadow modulation layers. The contiguous landscape geometry was split into separate sprites. As an authoring pre-process, the geometry along the split boundaries was expanded to allow for small separation of the warped sprites (the “seam” artifact.) This is similar to the expansion of the geometry along the split used by [Shade96]. More work is needed for automatic determination of the geometric expansion and better blending along the split boundaries.

The resource-use graph in Figure 22 shows three types of regulation. Simple triple framing, in which a frame is rendered and then

¹² In the longer film from which the barnyard sequence is taken, many of the shots have fixed cameras. In these shots, only the main characters need to be updated, so the performance gain is extremely high. This is similar to the time-honored technique of saving the z-buffer for fixed camera shots.

held for three frames, requires the most resources. Interpolated triple-framing requires the same amount of rendering resources, but interpolates through time using the warp described in Section 4.1 – the sprites are still rendered in lock-step but track the underlying characteristic geometry between renderings. The rest of the curves show threshold regulation with increasing geometric error threshold, from 0.1-0.8 pixels – the sprites are updated heterogeneously when the geometric error threshold is exceeded. The graph is normalized to the resource use of triple-framing.

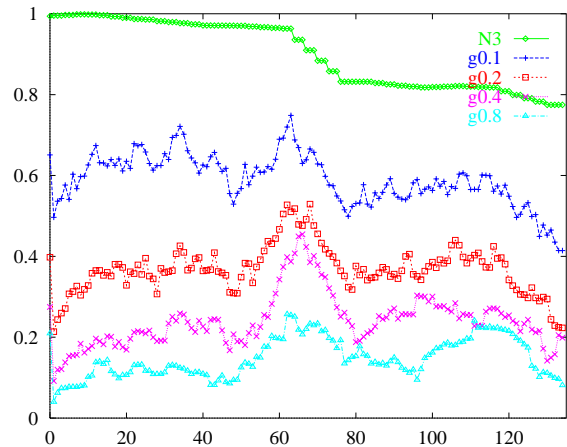


Figure 22: BARNYARD RESOURCE USE shows polygon counts as a 15-frame moving average. Pixel fill resource use is analogous. The top line is the triple-frame rendering. The lines below use threshold-regulation with increasing geometric threshold. As expected, as the pixel error tolerance goes up, the resource use goes down.

In the resulting animations, the most dramatic improvement in quality comes when the interpolation is turned on. The rest of the animations are nearly indistinguishable and use a fraction of the resources by rendering only those sprites whose geometric error exceeds the threshold.

Figure 23 shows the average pixel error for the same sequences shown in Figure 22. Each of the threshold-regulation sequences uses an error threshold smaller than the maximum error observed when triple-framing. Note that threshold-regulation is not the typical case and is shown here simply to demonstrate the performance advantage over the traditional graphics pipeline. Typically, all of the rendering resources are used to make the picture as good as possible.

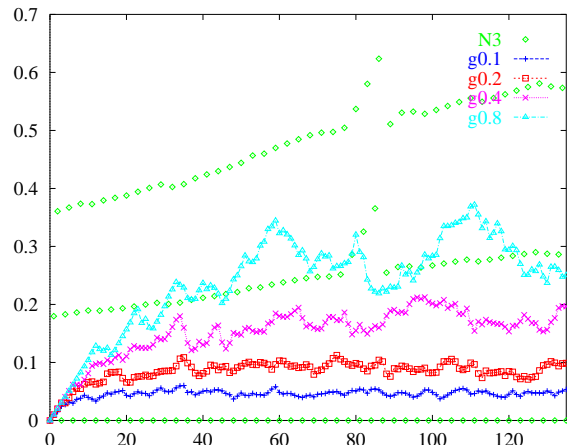


Figure 23: BARNYARD PIXEL ERROR shows average sprite pixel error per frame. Note that the triple-frame error is a saw-tooth that starts at 0, jumps to 1/2, and then jumps to full error value. The other curves oscillate below the given geometric threshold value.

8 Conclusions and Future Work

3D scenes should be factored into layers, with each layer having the proper sampling rates in both space and time to exploit the coherence of its underlying geometry and shading. By regulating rendering parameters using feedback from geometric, photometric, visibility, and sampling fiducials, rendering resources are applied where most beneficial. When not re-rendered, image warping suffices to approximate 3D motion of coherently factored layers. An affine warp provides a simple but effective interpolation primitive. These ideas yield 3-10 times improvement in rendering performance with the Talisman architecture with minor artifacts; greater performance can be controllably obtained with further sacrifices in fidelity.

Perceptual discontinuities may occur when a sprite's image is updated. Approximation with image warps captures the in-plane rotation, scale, and translation of an object, but not the out-of-plane rotation. The sprite image updates are sometimes perceived as a "clicking" or "popping" discontinuity. As the demand for higher quality 3D graphics increases display refresh rates, such artifacts will wane even at large factors of rendering amplification. More work is needed on the "seam" artifact (handling the boundaries of contiguous geometry placed in separate sprites.) Better modeling of the perceptual effects of regulation parameters is another area of future work [Horvitz97].

Factoring of shading terms is currently done using a fixed shading model that targets only the addition and over operations provided by hardware. Compilation of programmable shaders into layerable terms is an important extension. Many shading expressions, such as the shadow multiplication described in the appendix, can only be approximated with the over operator. We are interested in extending the system to target a fuller set of the image compositing operations.

Acknowledgements

The authors would like to thank the Microsoft Talisman Group and Microsoft Research, especially Jim Kajiya, Larry Ockene, Mark Kenworthy, Mike Toelle, Kent Griffin, Conal Elliott, Brian Guenter, Hugues Hoppe, Eric Horvitz, David Jenner, Andrew Glassner, Bobby Bodenheimer, Joe Chauvin, Howard Good, Mikey Wetzel, Susan O'Donnell, Mike Anderson, Jim Blinn, Steve Gabriel, Dan Ling, and Jay Torborg. Thanks to Nathan Myhrvold for the research direction and core ideas, and to Russell Schick for initial work on error-based sprite re-rendering. Thanks also to Allison Hart Lengyel and Julia Yang-Snyder.

Appendix: Shadow Sprites

For a fast-moving shadow on a slow-moving receiver, we update only the fast-moving shadow and use the compositor to compute the shadow modulation. Since the compositor supports only 'over', we use the following approximation.

Let $\mathbf{B}=[\beta\mathbf{B},\beta]$ be the receiver, where \mathbf{B} is the color and β is the coverage. Let $\mathbf{A}=[\alpha\mathbf{A},\alpha]$ be the desired shadow sprite, where \mathbf{A} is the color and α is the coverage. The compositor computes

$$\mathbf{A} \text{ over } \mathbf{B} = [\alpha\mathbf{A} + (1-\alpha)\beta\mathbf{B}, \alpha + (1-\alpha)\beta].$$

Let s be the shadow modulation obtained by scan-converting the geometry of the background while looking up values in the shadow map of the fast moving object, where 0 means fully in shadow and 1 means fully illuminated.

The desired result is $\mathbf{C}' = [s\beta\mathbf{B}, \beta]$. By letting $\mathbf{A} = [0, (1-s)\beta]$, we get $\mathbf{C}' = \mathbf{A} \text{ over } \mathbf{B}$, or $\mathbf{C}' = [s\beta\mathbf{B} + (1-s)(1-\beta)\beta\mathbf{B}, \beta + (1-s)(1-\beta)\beta]$ which is close to the correct answer. Where there is no shadow, s is 1 and we get the correct answer of $[\beta\mathbf{B}, \beta]$. Where coverage is complete, β is 1 and we get the correct answer of $[s\mathbf{B}, 1]$. The problem lies in regions of shadow and partial coverage.

Ideally, the shadow modulation needs a color multiply operator '*' defined by $s*[\beta\mathbf{B}, \beta] = [s\beta\mathbf{B}, \beta]$. This is a per-pixel version of the Porter-Duff 'darken' operator. Note that this operator is not associative, and so requires the saving of partial results when used in a nested expression.

References

- [Blinn96] Consider the Lowly 2x2 Matrix, Jim Blinn, *IEEE Computer Graphics and Applications*, March 1996, pp. 82-88.
- [Chen93] View Interpolation for Image Synthesis, Shenchang Eric Chen and Lance Williams, *SIGGRAPH 93*, pp. 279-288.
- [Chen95] QuickTime VR - An Image-Based Approach to Virtual Environment Navigation, Shenchang Eric Chen, *SIGGRAPH 95*, pp. 29-38.
- [Cook84] Shade Trees, Robert L. Cook, *SIGGRAPH 94*, pp. 223-232.
- [Dorsey95] Interactive Design of Complex Time Dependent Lighting, Julie Dorsey, Jim Arvo, Donald P. Greenberg, *IEEE Computer Graphics and Applications*, March 1995, Volume 15, Number 2, pp. 26-36.
- [Funkhouser93] Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments, Thomas A. Funkhouser and Carlo H. Séquin, *SIGGRAPH 93*, pp. 247-254.
- [Gill81] Practical Optimization, Philip E. Gill, Walter Murray, and Margaret H. Wright, Academic Press, London, 1981.
- [Greene93] Hierarchical Z-Buffer Visibility, Ned Greene, Michael Kass, Gavin Miller, *SIGGRAPH 93*, pp. 231-238.
- [Greene94] Error-Bounded Antialiased Rendering of Complex Environments, Ned Greene and Michael Kass, *SIGGRAPH 94*, pp. 59-66.
- [Guenter95] Specializing Shaders, Brian Guenter, Todd B. Knoblock, and Erik Ruf, *SIGGRAPH 95*, pp. 343-350.
- [Hanrahan90] A Language for Shading and Lighting Calculations, Pat Hanrahan and Jim Lawson, *SIGGRAPH 90*, pp. 289-298.
- [Hofmann89] The Calculus of the Non-Exact Perspective Projection, Scene-Shifting for Computer Animation. Georg Rainer Hofmann. *Tutorial Notes for Computer Animation, Eurographics '89*.
- [Horvitz96] Flexible Rendering of 3D Graphics Under Varying Resources: Issues and Directions, Eric Horvitz and Jed Lengyel, In *Symposium on Flexible Computation*, AAAI Notes FS-96-06, pp. 81-88. Also available as Microsoft Technical Report MSR-TR-96-18.
- [Horvitz97] Decision-Theoretic Regulation of Graphics Rendering, Eric Horvitz and Jed Lengyel, In *Thirteenth Conference on Uncertainty in Artificial Intelligence*, D. Geiger and P. Shenoy, eds., August 1997.
- [Hubschman81] Frame to Frame Coherence and the Hidden Surface Computation: Constraints for a Convex World, Harold Hubschman, and Steven W. Zucker, *SIGGRAPH 81*, pp. 45-54.
- [Kay86] Ray Tracing Complex Scenes, Timothy L. Kay and James T. Kajiya, *SIGGRAPH 86*, pp. 269-278.
- [Maciel95] Visual Navigation of Large Environments Using Textured Clusters, Paolo W. C. Maciel and Peter Shirley, *Proceedings 1995 Symposium on Interactive 3D Graphics*, April 1995, pp. 95-102.
- [Mark97] Post-Rendering 3D Warping, William R. Mark, Leonard McMillan, and Gary Bishop, *Proceedings 1997 Symposium on Interactive 3D Graphics*, April 1997, pp. 7-16.
- [Meier96] Painterly Rendering for Animation, Barbara J. Meier, *SIGGRAPH 96*, pp. 477-484.
- [Molnar92] PixelFlow: High-Speed Rendering Using Image Composition, Steve Molnar, John Eyles, and John Poulton, *SIGGRAPH 92*, pp. 231-240.
- [McMillan95] Plenoptic Modeling: An Image-Based Rendering System, Leonard McMillan, Gary Bishop, *SIGGRAPH 95*, pp. 39-46.
- [Porter84] Compositing Digital Images, Thomas Porter and Tom Duff, *SIGGRAPH 84*, pp. 253-259.
- [Regan94] Priority Rendering with a Virtual Reality Address Recalculation Pipeline, Matthew Regan and Ronald Pose, *SIGGRAPH 94*, pp. 155-162.
- [Segal92] Fast Shadows and Lighting Effects Using Texture Mapping, Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, Paul Haeberli, *SIGGRAPH 92*, pp. 249-252.
- [Schaufler96a] Exploiting Frame to Frame Coherence in a Virtual Reality System, Gernot Schaufler, *Proceedings of VRAIS '96*, April 1996, pp. 95-102.
- [Schaufler96b] A Three Dimensional Image Cache for Virtual Reality, Gernot Schaufler and Wolfgang Stürzlinger, *Proceedings of Eurographics '96*, August 1996, pp. 227-235.
- [Shade96] Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments, Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, John Snyder, *SIGGRAPH 96*, pp. 75-82.
- [Shelley82] Path Specification and Path Coherence, Kim L. Shelley and Donald P. Greenberg, *SIGGRAPH 82*, pp. 157-166.
- [Snyder97] Visibility Sorting for Dynamic, Aggregate Geometry, John Snyder, Microsoft Technical Report, MSR-TR-97-11.
- [Torborg96] Talisman: Commodity Real-time 3D Graphics for the PC, Jay Torborg, Jim Kajiya, *SIGGRAPH 96*, pp. 353-364.
- [Xie95] Feature Matching and Affine Transformation for 2D Cell Animation, Ming Xie, *Visual Computer*, Volume 11, 1995, pp. 419-428.

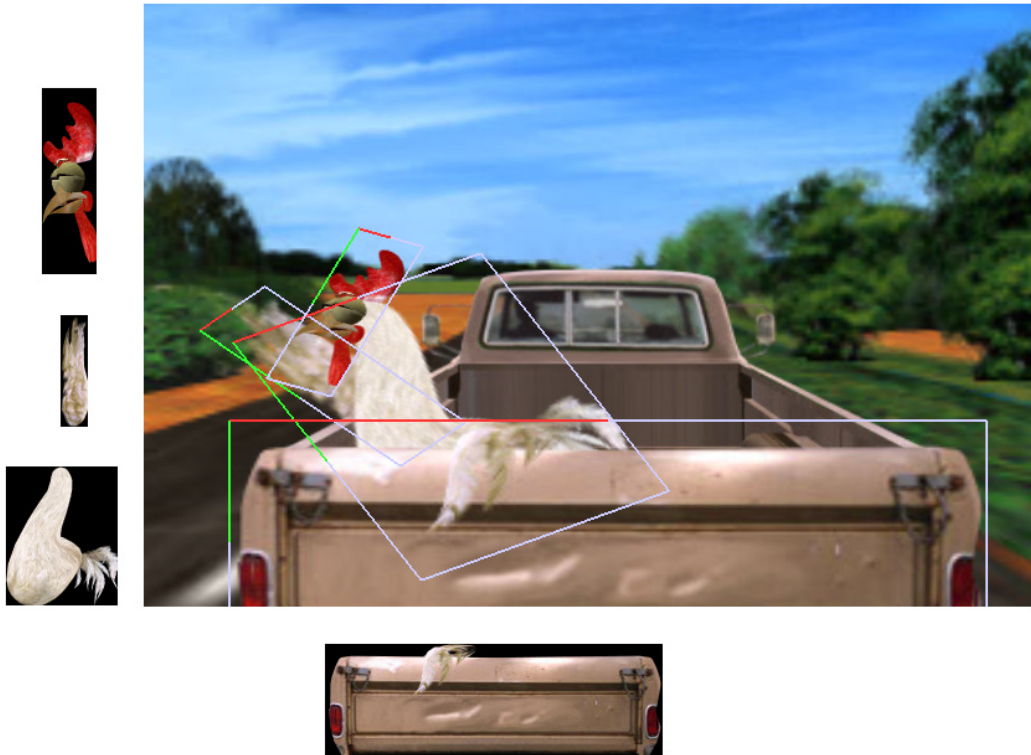


Figure 24: CHICKEN CROSSING sequence used 80 layers, some of which are shown separately (left and bottom) and displayed in the final frame with colored boundaries (middle). The sprite sizes reflect their actual rendered resolutions relative to the final frame. The rest of the sprites (not shown separately) were rendered at 40-50% of their display resolution. Since the chicken wing forms an occlusion cycle with the tailgate, the two were placed in a single sprite (bottom).



Figure 25: CANYON FLYBY used 10 layers with a geometric fiducial threshold of 4 pixels. The average sprite update rate was 19% with little loss of fidelity.



Figure 26: BARNYARD was factored into 119 geometry layers, 2 shadow map layers, and 2 shadow modulation layers. Threshold regulation for various geometric fiducial thresholds is compared in Figures 22 and 23.