Streaming 3D with The Irregular Back Buffer (IBB) The Presentation Renderer and Per Triangle Surface Data (PTSD)

We propose a change to the graphics pipeline, enabling streaming 3D along with simplified client/display systems, more-uniform performance, ease of authoring, etc. We introduce the "Irregular Back Buffer" (IBB): the collection of potentially visible triangles and their shaded results. We also introduce "Per Triangle Surface Data"(PTSD), a new way of storing data over triangles.

Motivation

Everyone creates videos. Few create 3D applications. We propose a change to the 3D pipeline to lower the barriers to create and deliver 3D more like video.

Real time 3D is significantly more complex than video. They both get pixels to the screen. But a video is always the same, while 3D is generated on the fly. Video is "just data" that plays everywhere from the latest gaming console to a 20-year-old phone. 3D is an executable, and a database of art assets. These are tightly connected with the target platform, with different platforms forming separate markets.

The video user experience is fundamentally different from 3D. Video applications present a wall of choices that play immediately when selected. 3D Games, in stark contrast, usually require a long time to download the executable and data. System updates are also often required – new drivers, etc.

Real time 3D generates images on the client; image quality is constrained by the client's capabilities – a high-end PC can generate dramatically better results than a mobile phone. This affects the structure of the market, as only a small group can afford the high-end.

Streaming 3D on top of video is an alternative to our proposal: send user input to the server, render the frames on the server, compress as digital video, and send to the client. This is challenging for real time 3D, with critical latency requirements. New images are needed very quickly as the view changes in response to user input. This is particularly challenging for VR and AR as the latency requirements are very short yet critical.

We separate the part that generates the final view from the part that computes the expensive lighting and shading. The main renderer generates data that, like video, is "just data". And, the Presentation Renderer uses that data to generate the final view. This allows buffering on the client where the performance-critical bit is performed.

The goal is to decouple the client from the server. Like video, the client just needs the data; it doesn't care how it's generated. This facilitates a more attractive content market, as content creators no longer need to narrowly target a specific platform.

Introduction

The IBB holds lighting and shading results. The Presentation Renderer generates novel views from these results. Eg when the VR HMD moves. It decouples computing the expensive lighting and shading from the inexpensive rendering the final view.

- Reduce/eliminate latency dependencies. Stream and buffer. The final view is generated on the client.
- Exploit temporal coherency in surface space. Screen space changes dramatically every frame, but surface space is more stable (often constant)
- Share IBB results for the left and right eye
- Update the IBB only every nth frame (e.g., display at 240 FPS, but light/shade at 24 FPS)
- Update at different rates. (E.g., update hero's face every frame)
- Accumulate results over n frames (e.g., per-frame jitter)
- AI generation, upsampling, etc in surface space.
- Streaming. The IBB can be generated somewhere else and somewhen else and streamed to the client.
- Efficient Multicasting – generate IBB data once, and share with many clients.
- Streaming business models. Pick experience from menu. Immediately immersed. Ad supported. Etc.
- Standardize the IBB format, enabling innovation for both producers and consumers.
- Similar to video: "Just data". Minimizes platform dependencies, like executables and shaders.
- AI target/output format. What will AI 3D output look like? A PS5 executable? A Vulkan app for Meta Quest 3?

Back Buffers and Front Buffers

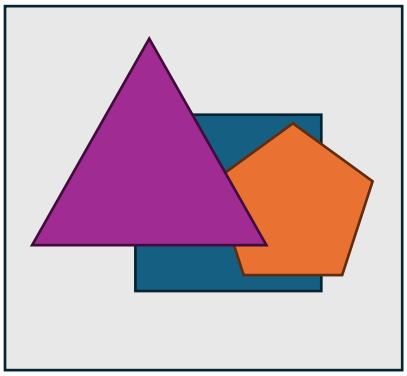
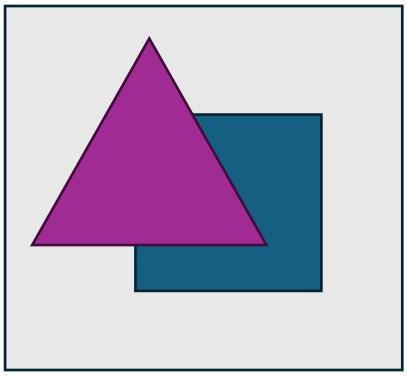
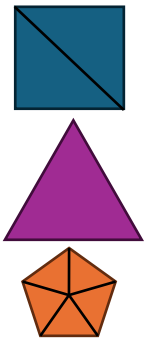


Figure 1 Front Buffer (displayed)



Back Buffer (in progress)



PTSD Irregular Back Buffer

Realtime renderers typically maintain two pixel-buffers[i]. The "front buffer" is actively displayed, while the "back buffer" is hidden. The back buffer holds the partial results as rendering creates them. It's a process that takes time. So, we construct the back buffer out of sight, while the user sees the front buffer.

Renderers typically redraw the entire back buffer every frame. Clear it, and several other buffers (e.g., Shadow Buffers, G Buffer, …). Reconstruct it from scratch. This is required because the image changes as the view changes. E.g., even though the grass is still green, it moved to different pixels when the player turned their head. It's constant in a sense (in surface space), but any individual pixel changes. PTSD exploits this, and stores in surface space instead of screen space. Note: several techniques reuse results from previous frames to improve image quality. In contrast, IBB stores information in surface space, requiring fundamentally less updating. Many useful cases can even use constant data – like a real estate listing.

The front and back buffers are 2D arrays/rectangles (i.e., "texture", "render target", …). The back buffer is displayed by swapping it with the front buffer. The front becomes the back and the back becomes the front. Copying is notably an option, with conversions or simple transformations possible enroute.

Presentation Renderer

The IBB is also a 2D surface, but composed of triangles to form arbitrary shape. The camera can view this shape from different points of view. We accomplish this with the Presentation Renderer. It "swaps" by rasterizes the PTSD triangles to the front buffer (or to the "real" back buffer, which then gets swapped the old fashioned way). PTSD enables the Presentation Renderer to be efficient. It's effectively a single texture.

Note the Presentation Renderer can render many times from the same PTSD data. A scene with constant shading and lighting could use constant PTSD data. The renderer could maintain more than one set of PTSD; e.g., blend between multiple lighting conditions. Though it's important to note that view-dependent lighting (e.g., reflections) isn't a simple constant in surface space. It includes the view direction so requires e.g., lenticular representation, dynamic updates, spherical harmonics, Gaussians, AI solvers, etc.

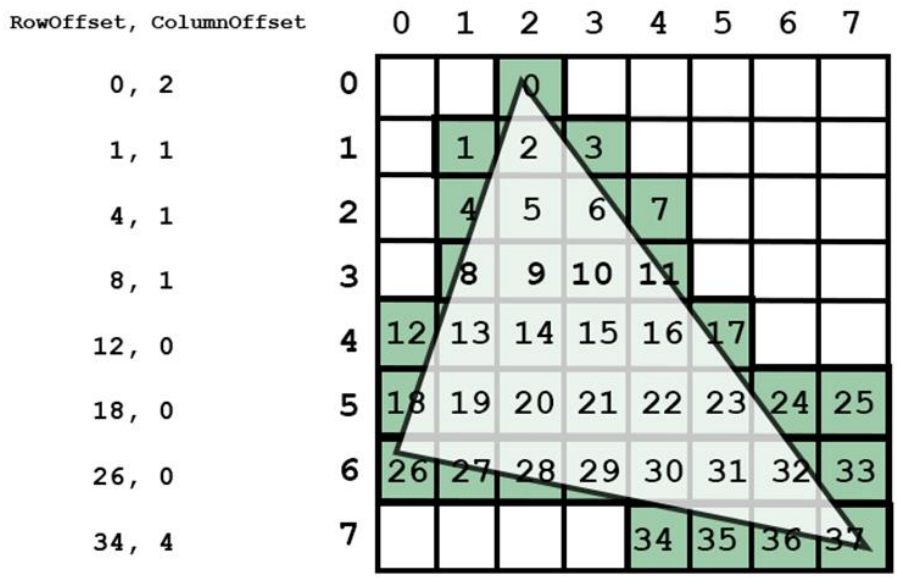
Per Triangle Surface Data

PTSD is an alternative to texture mapping. A texture map is a 2D rectangle/array of texels. Artists and tools use texture coordinates to map between the surface of the triangle and the texture map. Given a location on a triangle's surface, we compute the memory addresses where the corresponding texels are stored.

This address calculation is trivial for 2D arrays/textures. It's less trivial for PTSD, though still straightforward. The advantages of PTSD are:

- Not constrained to a rectangle
- Stores only the required texels - not empty space
- Like a huge texture atlas, but we don't need to care about efficient use of the rectangle space
- Variable-resolution. Could extend such that triangles could have different texel density – including uniform color.
- Can reuse same data for multiple triangles
- Can use different bit depth, color space, compression schemes, etc. for different triangles.

PTSD can be used wherever textures are used, including as a render target. We focus here on use as the IBB. But, other uses are also interesting (e.g., an irregular G Buffer to support dynamic lighting)



This is the PTSD representation for a single triangle. The 38 darker texels are what we store. `RowOffset` and `ColumnOffset` are our map for computing which texel we need, given a row and column.

The simple rectangle case is trivial. Given a rectangle that's `w` texels wide, the address for texel (`x, y`) is given by:

```
addr = y*w + x.
```

We compute the address for a PTSD triangle:

```
Addr = RowOffset[y] + x – ColumnOffset[y]
```

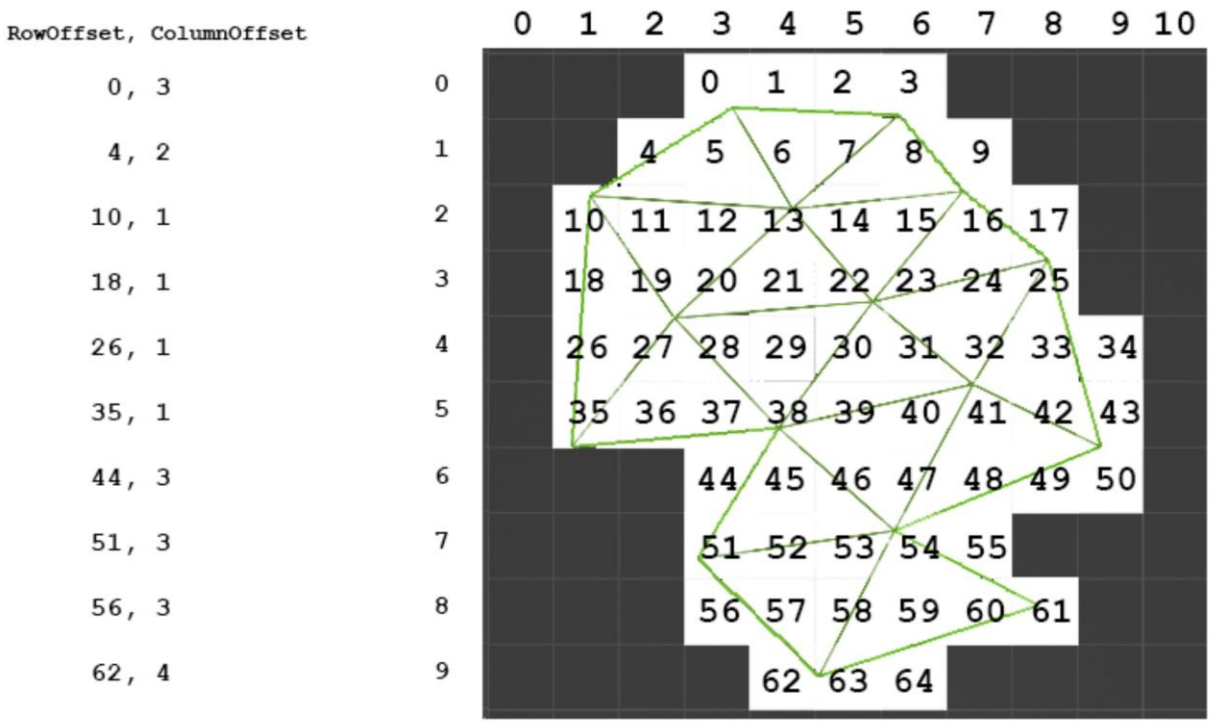
For example: the texel at (5,7) at address:

```
RowOffset[y] + x – ColumnOffset[y]

34 + 5 – 4 = 35
```

Its important to note this format preserves the texel orientations, relative to the triangle. This is critical for ensuring seamless results where triangles share an edge.

Note RowOffset and ColumnOffset are local to the triangle. An additional index is required to support multiple triangles – the address of the first texel in the triangle, for each triangle.



This is an example of PTSD representing many triangles. This improves efficiency by storing shared texels only once. Note that perimeter texels might be shared with triangles that aren't included in the same PTSD chunk.

Note address calculation remains the same but all of these triangles have the same base offset. RowOffset and ColumnOffset are valid for all triangles in the meshlet/patch.

---

[i] More than one back buffer is possible, e.g a circular queue of 3 buffers