Shadow Techniques from Final Fantasy XVI

Sammy Fatnassi

July 31, 2023

Revision 1

http://www.jp.square-enix.com/tech/publications.html

Contents

1 Introduction

Final Fantasy is a renowned video game franchise with riveting stories accompanied by incredible graphics, each iteration pushing forward the expectation of what is possible. In this paper, I present implementation details of high-quality shadow generation in the context of real-time rendering in *Final Fantasy XVI*. This document covers only a subset of techniques novel enough to share. The production of any game is a colossal team effort and in respect to this *'our'* is used when referring to the technology and *'I'* for the author.

In sections 2 and 3, I present our *Tiled Deferred Shadow* renderer that reduces GPU computing cost and improves the quality of character shadows when paired with iteratively composited *High-Quality Shadows* over the previous *Tiled Deferred Shadow* results. In section 4, I discuss a new method named *Oriented Depth Bias* to solve issues associated with the use of *Shadowmap*. This simple approach handles self shadow issues (*shadow acne*) and is effective in replacing the hardware depth bias traditionally used. In section 5, I present visualizers for the inspection of results and diagnosing problems that indirect computing of shadows makes arduous to investigate otherwise. Section 6 close with some conclusions.

2 Deferred Shadows

The *Tiled Deferred Lighting* technique made possible by programmable GPU shaders, introduced a paradigm shift in real-time rendering. It allowed splitting surface parameter computation from lighting calculations and lead to improved performance at the cost of higher memory usage. This is achieved by storing surface parameters in *Geometry Buffers* (known as GBuffers) as a first pass, deferring the costly lighting computation to a second pass limited to the visible pixels. It has since been adopted by most games for their lighting solution, including *Final Fantasy XVI*.

In *'A Scalable Real-Time Many-Shadowed-Light Rendering System'* [2], Li Bo introduced the '*Tiled Deferred Shadow*' technique, further splitting the shadow from the lighting computation and resulting in performance improvement by reducing *VGPR Pressure* in the lighting pass. By extending this technique, we were able to additionally support an arbitrary number of high-quality shadows on object/light pairs, thanks to the addition of our *Closeup Shadowmap* technique. The memory overhead remained low and fixed no matter the number of *Closeup Shadowmaps* by rendering their *Shadowmap* iteratively and re-using the same render target.

A rudimentary *Tiled Deferred Shadow* system requires little implementation effort when already using a *Tiled Deferred Lighting* renderer. It can be achieved by moving the *per pixel visibility* computation of each light from the *Tiled Deferred Lighting* pass to a new *Tiled Deferred Shadow* pass and storing intermediate results in buffers that are read back by the *Tiled Deferred Lighting* pass. However, this naïve approach increases the memory budget to unrealistic levels when expecting support for many lights per pixels. This section goes over our *Tiled Deferred Shadow* implementation details that achieved better performance and memory reduction *(see figure 1 for a quick overview)*. The first phase runs on the CPU and initiates the separates GPU dispatches of the following *Tiled Deferred Shadow* phases.

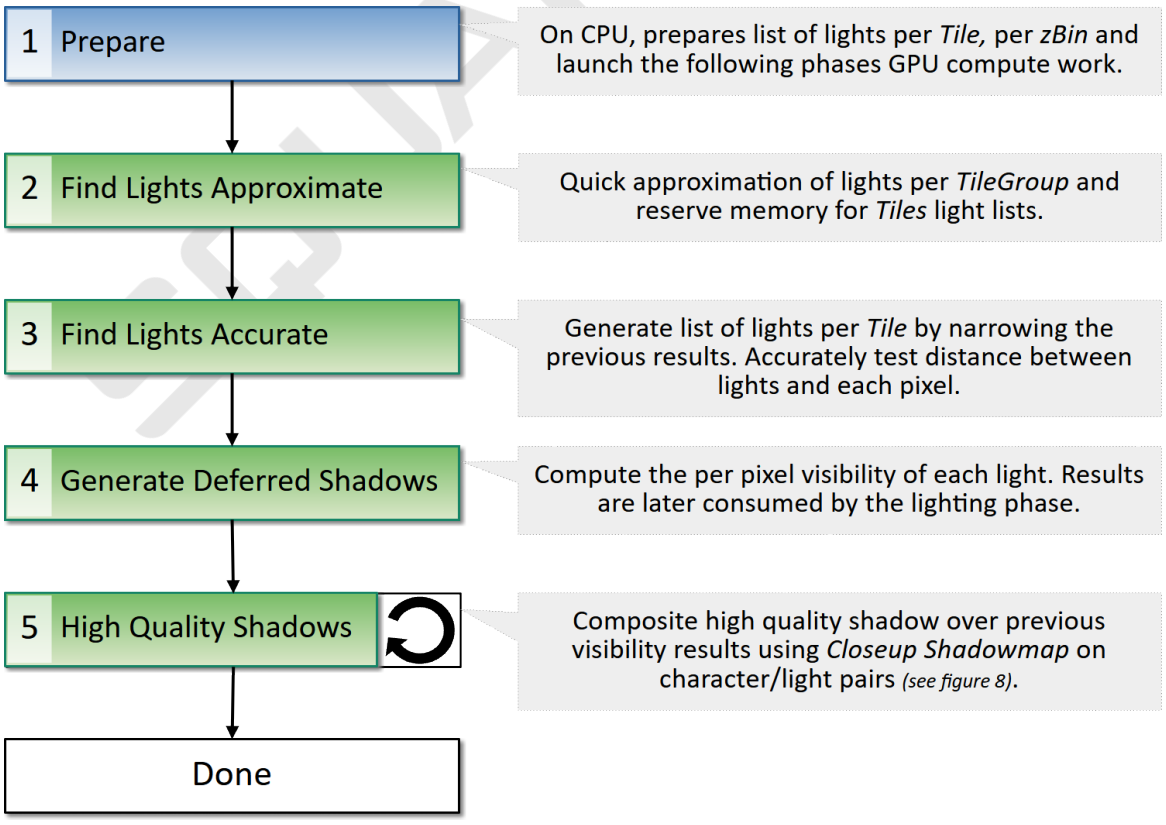


Figure 1: *Tiled Deferred Shadow* flow

2.1 Phase: Prepare

In this initial phase, multiple lists of lights are generated on the CPU to be used as buffer inputs in the following phases accelerating the shadow processing. It is the only step not processed on the GPU.

2.1.1 Output: *zBin Buffer*

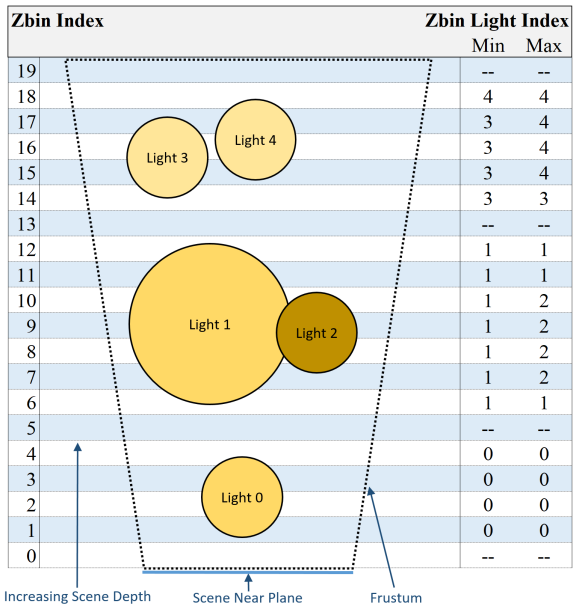
This buffer is used to quickly reject lights outside a depth distance *(see 'Clustered Deferred and Forward Shading' [4] for more information)*. We discretize our scene view depth range in 1024 intervals called *zBin* and store a list of valid lights for each one. By first sorting the lights by their closest distance to the scene view near plane, storage requirements are greatly reduced by only keeping the minimum and maximum light index of each *zBin* (*see figure 2a*).

Figure 2a shows a sample of a scene view discretized in 20 *zBins*. For any single depth value, the associated *zBin* is selected to know that potentially valid lights are between *min* and *max* (inclusively). For example, with a depth value in *zBin* 10, lights indices between 1 and 2 are found to be the only potentially valid ones.

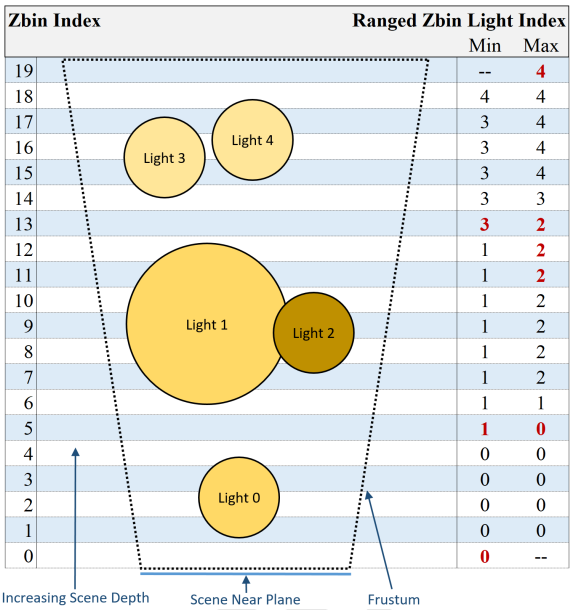
Using *zBins* works well for *single-depth* values but falls short when used to find lights inside a *depth range*. Simply taking the minimum and maximum light indices of the first and last *zBins* does not guarantee valid results (*see figures 3a and 3b for error examples*). Iterating every *zBins* in a depth range solves this but without the speed benefits of finding the minimum and maximum light indices with only two lookups. Our solution to this is to generate a second *zBin* dataset named *zBin (ranged)* and fill it with values handling depth ranges (*see figure 2b*).

- For each *zBin* 'min light index', we save lowest value between the current and last *zBin*.
- For each *zBin* 'max light index', we save highest value between the current and first *zBin*.

Since the *zBin (ranged)* results are more conservative than the standard *zBin* ones (more false positives), both datasets are stored in the *zBin Buffer*. With depth values inside 1 or 2 contiguous *zBins*, we use the standard *zBin* buffer and the *zBin (ranged)* buffer otherwise.

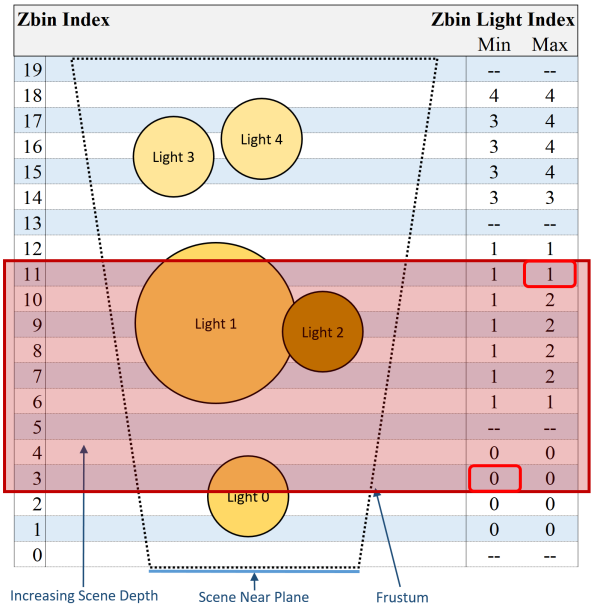


(a) **Standard** ***zBin***: for any single scene view depth, we find the associated *zBin* to know that potentially valid lights are between *min* and *max* (inclusively). For example, for a depth value in *zBin* 10, we find that the potentially valid light indices are between 1 and 2.

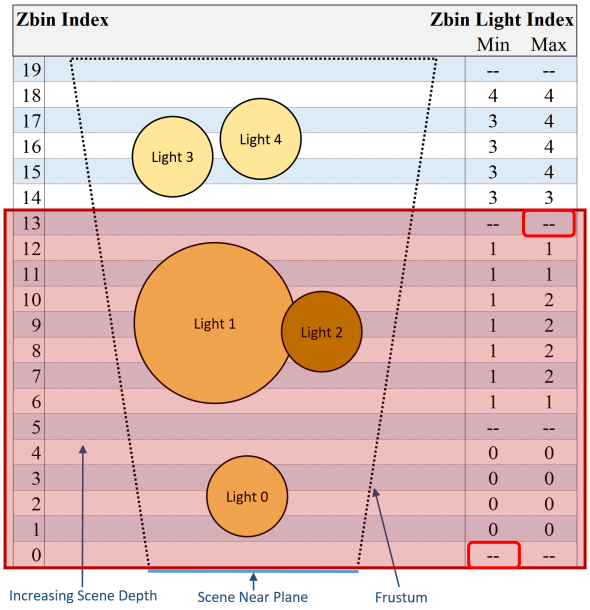


(b) **Ranged** ***zBin***: for a scene view depth range, we find the associated first and last *zBins* to know that potentially valid lights are between *first zBin min* and *last zBin max* (inclusively). For example, for depth values starting in *zBin* 2 and ending in *zBin* 13, we find that the potentially valid light indices are between 0 and 2.

Figure 2: Scene view discretized in 20 *zBins* with light indices sorted by their closest distance to the *scene near plane*.



(a) In *zBin* range [3,11] we find light indices [0,1], missing index 2.



(b) In *zBin* range [0,13] we find no light index, missing indices 0,1,2 entirely.

Figure 3: Error examples when relying on the first and last *zBin* entries of a depth range to find the valid light indices.

2.1.2 Output: *Tile Light Mask Buffer*

This buffer is used to quickly discard lights outside a *Tile*'s screen area. To reduce the memory requirement of supporting 1024 lights per *Tile* while keeping a predictable direct access to results, a 32 Bytes bit-field is used with an additional 4 Bytes bit-field for skipping a group of 32 consecutive invalids' lights (*see figure 4*).

While the *Shadow Tile* size used by the *Tiled Deferred Shadow* is set to 8x8 pixels, we keep the CPU computation cost of this buffer down by using larger *Tiles* of 32x32 pixels (*see figure 5*). In 4K resolution, this translates to only processing 8,100 *Tiles*, down from the original 129,600, at the cost of a slight increase in false positives discarded in the next phase (*see section 2.2*).

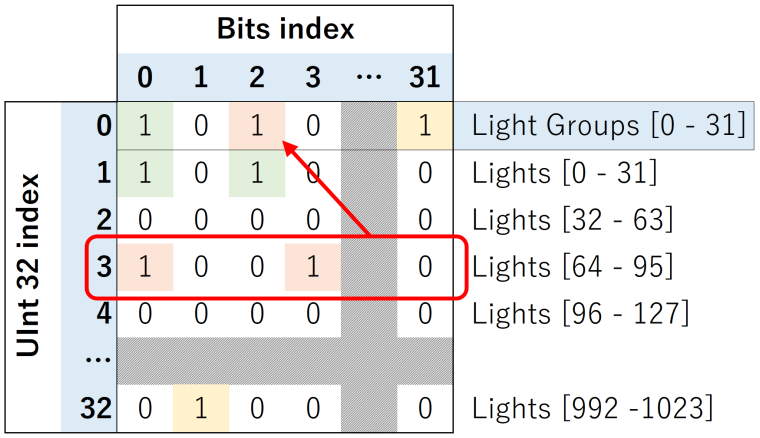


Figure 4: ***Light Mask Bit-Fields Format***: Each of the 1024 lights has 1 bit to store its validity using the light index as an offset in the last 32 Bytes of the associated *Tile Light Mask Buffer* entry. Additionally, groups of 32 consecutive lights are formed and use 1 bit in the first 4 bytes of the *Tile Light Mask Buffer* to indicate when they have at least one valid light. We use this to accelerate iteration over the lights by ignoring empty ranges. In this example, *Light Groups* 0, 2 and 31 are marked active because they each have at least 1 associated valid light.

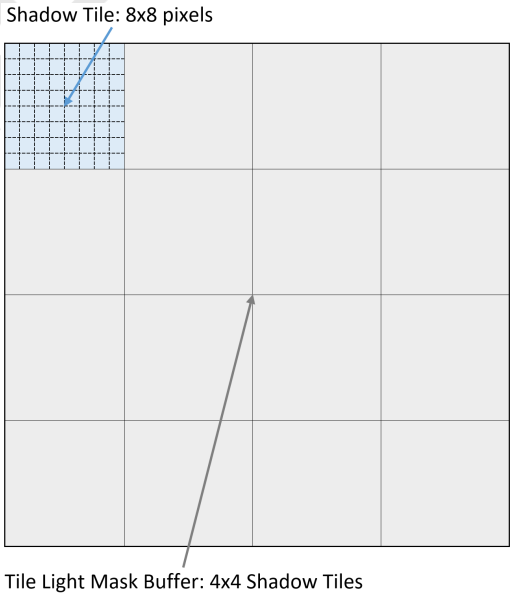


Figure 5: Comparison of a *Shadow Tile* used as a *Compute Work Unit* size on GPU and a *Tile Light Mask Buffer* entry's area size when storing the validity of lights per screen region.

2.2 Phase: Find Lights Approximate

This second phase has two objectives: finding the potentially valid lights per *Shadow Tile* and reserving memory for storage of light lists built in the following phases. We operate in *Compute Work Units* of 8x8 and keep this phase inexpensive by evaluating 1 *Shadow Tile* per thread instead of processing per pixel results. We prioritize speed over accuracy with some false positives expected that the next phase will filter out.

First, each *Shadow Tile* minimum and maximum scene depths are fetched from the *Hierarchical Depth Buffer* [1] to generate a *Froxel* by combining them with the screen coordinates. Next, the *Froxel* valid light indices are found by keeping the lights present in both the corresponding *Tile Light Mask Buffer* and the *zBin Buffer* entries. We finalize vetting each light using bound intersections with the *Froxel* using a quick sphere/sphere test, a cone/sphere test taken from the article 'Cull That Cone!' [1], and an axis aligned bounding box (AABB) test by bringing the voxel into the light clipping space (*see listing 2 for more details*).

2.2.1 Output: *Tile Group Light Mask Buffer*

Not to be confused with the previously introduced *Tile Light Mask Buffer*, this buffer stores the valid lights approximation per *Tile Group*. To reduce the memory usage, we use one *Light Mask Entry* per 8x8 *Shadow Tiles*, using the format introduced in *figure* 4. In 4K resolution, this brings the memory usage from ~4.4MB to ~0.3MB. By using *Compute Work Units* matching the *Tile Group* size, we leverage fast *Lane Operators* in the computation to avoid slow memory exchange between threads.

2.2.2 Output: *Tile Light SlotID Buffer*

This buffer stores the *Light SlotID* of each *Shadow Tile* after computation. Few *Shadow Tiles* contain a high number of lights and allocating the same fixed amount of memory for all to handle the maximum count is wasteful. Instead, a flexible memory approach is used where each *Shadow Tile* reserves enough memory in a shared buffer to handles their approximated light count. This allows support for a greater number of lights per *Shadow Tile* while reducing the memory consumption of the two light lists generated in the following phases. With this approach, the associated memory usage went from ~22.1MB to ~5.5MB.

To assign a *Light SlotID* per *Shadow Tile*, one *Compute Shader Thread* per *Compute Work Unit* first sums up the number of valid lights in the entire *Tile Group*. Next, it reserve memory for these entries with a single *atomic increment* on a shared global counter[2]. Finally, each thread computes its own *Shadow Tile*'s *Light SlotID* by offsetting the atomic value by the number of entries used by previous *Tiles* in the *Tile Group*. This computation relies on *lane operations* as much as possible for a speed boost when accessing neighboring thread's values.

Note: It is helpful to also support a debug mode without the flexible memory approach, allocating space for a fixed number of lights per *Shadow Tile* instead. Because the *Light SlotID* values are now predictable (no *a priori* needed besides the *Shadow Tile* coordinates) it becomes easier to inspect the *Shadow Tile* light lists produced in the next steps.

[1] When detecting a maximum scene depth matching the scene view far plane (skybox's pixel for example), we read each pixel's scene depth individually to find the maximum valid depth. This substantially reduce the number of false positive lights caused.

[2] The single thread access greatly reduces atomic contention and improves performance.

```
globallycoherent RWByteAddressBuffer rw_SlotCounters;

//======================================================================
// Allocate a ShadowSlotID that will be used to store list of lights
uint AllocateShadowSlotID(in uint inTileIndex, in uint inTileLightCount)
{
  // Reserve fixed amount of memory for debuging with predictable location
  if( DEBUG_FIXED_LIGHTCOUNT > 0 ){
    return inTileIndex * RoundUp(DEBUG_FIXED_LIGHTCOUNT, 3);
  }

  // Round up to next multiple becase 3 indices stored per uint
  inTileLightCount       = RoundUp(inTileLightCount, 3);
  uint tileGrpLightCount = WaveActiveSum(tileLightCount);
  uint tileGrpSlotID     = 0;

  // Single thread allocates space for entire TileGroup
  if( WaveIsFirstLane() ){
    static const uint kShadowCounterOffset = 0;
    rw_SlotCounters.InterlockedAdd( kShadowCounterOffset,
                                    tileGrpLightCount,
                                    tileGrpSlotID);
  }

  // Tile get its SlotID by offsetting the starting location of the
  // Group reserved memory with the sum of previous Tile Slots count
  uint shadowSlotID  = WaveReadLaneFirst(tileGrpSlotID);
  shadowSlotID      += WavePrefixSum(inTileLightCount);
  return shadowSlotID;
}

//======================================================================
// Allocate a VisibilitySlotID that will be used to store the visibility
// of one light at each pixel of a Tile (1 slot = 8x8 pixels (64 bytes))
uint AllocateVisibilitySlotID()
{
  // Single thread allocates space for entire Tile
  uint slotID;
  if( WaveIsFirstLane() ){
    static const uint kVisibilityCounterOffset = 4;
    rw_SlotCounters.InterlockedAdd(kVisibilityCounterOffset, 1, slotID);
  }

  uint visibilitySlotID = WaveReadLaneFirst(slotID);
  return visibilitySlotID;
}
```

Listing 1: 'SlotID allocator' compute shader pseudo code

```
RWByteAddressBuffer  rw_TileShadowSlotIds;
RWByteAddressBuffer  rw_TileGroupLightMask;

// Work unit outputs 1 TileGroup (8x8 Tiles), each thread evaluate 1 Tile
[numthreads(TILE_LIGHTGROUP_SIZE, TILE_LIGHTGROUP_SIZE, 1)]
void CS_Phase2_FindLightsApprox( uint2 inTileCoord : SV_DispatchThreadID )
{
  bool isValidTile = all(inTileCoord < cCommon.m_TileResolution.xy);
  if( isValidTile ){
    const uint tileIdx         = GetTileIndex(inTileCoord);
    const float2 tileNearFar   = GetTileNearFarDepth(inTileCoord);
    const int2 zBinIndexMinMax = GetZBinIndexFromDepth(tileNearFar);
    const ZBin binRange        = GetRangeZBinFromIndex(zBinIndexMinMax);
    uint lightGroupMask    = GetTileLightGroupMask(inTileCoord);
    lightGroupMask         = IntersectLightGroup(binRange, lightGroupMask);
    uint tileLightCount    = 0;
    uint validLightGrpMask = 0;
    //-----------------------------------------------------------------
    // Process each valid group of 32 lights
    while (lightGroupMask) {
      uint validLightMask = 0;
      uint lightGrpBit    = firstbitlow(lightGroupMask);
      uint lightMask      = GetTileLightMask(lightGrpBit);
      lightMask           = IntersectLightMask(binRange, lightMask);
      //---------------------------------------------------------------
      // Process each valid light in current group of 32 lights
      while( lightMask ){
        uint lightBit       = firstbitlow(lightMask);
        uint lightIdx       = lightGrpBit*32 + lightBit;
        LightInfo lightInfo = GetLightInfo(lightIdx);
        // All intersection tests between 'Light/Froxel' here
        bool bValidLight    = LightFroxelTests(lightInfo, inTileCoord);
        lightMask          ^= 1u << lightBit; //Remove processed light
        if( bValidLight ){
          tileLightCount += 1;
          validLightMask |= 1 << lightBit;
        }
      }
      // Store light indices validity of current group with 32bits mask
      uint tileGroupLightMask = WaveActiveBitOr(validLightMask);
      if( WaveIsFirstLane() ){
        uint lightMaskAdr = GetLightMaskEntryAdress(tileIdx, lightGrpBit);
        rw_TileGroupLightMask.Store(lightMaskAdr, tileGroupLightMask);
      }
      //---------------------------------------------------------------
      // Detect if current 32 lights group is valid in any Tile of TileGroup
      validLightGrpMask |= tileGroupLightMask != 0 ? 1 << lightGrpBit : 0;
      lightGroupMask    ^= 1 << lightGrpBit; //Remove processed group
    }
    //-----------------------------------------------------------------
    // Store LightGroup mask entry for the TileGroup
    if( WaveIsFirstLane() ){
      uint lightMaskAdr = GetLightMaskGroupAdress(tileIdx);
      rw_TileGroupLightMask.Store(lightMaskAdr, validLightGrpMask);
    }
    // Allocate and store each Tile ShadowSlotIDs
    uint shadowSlotID  = AllocateShadowSlotID(tileIdx, tileLightCount);
    rw_TileShadowSlotIds.Store(tileIdx*4, shadowSlotID);
  }
}
```

Listing 2: 'Find Lights Approximate' compute shader pseudo code

2.3 Phase: Find Lights Accurate

This 3^{rd} phase filter light indices from the *Tile Group Light Mask Buffer* using more accurate per pixel tests on each *Shadow Tile*. Since previous results are shared by multiple *Shadow Tiles* and the *Froxel* tests create false positives (*see figure 6*) we eliminate as many as possible before computing the shadows in the next step. A list of lights per *Shadow Tile* is stored in an *Early Light List buffer* at a location calculated from the *Light SlotID*. The work is accomplished by a compute shader using a group size matching our *Shadow Tile* size of 8x8 pixels. With each *Compute* thread handling 1 pixel, we read the scene depth and iterate over the *Tile Group* lights found in the *Tile Group Light Mask Buffer* entry. The *Shadow Tiles* consider a light valid when it is in range of at least one of its pixels.

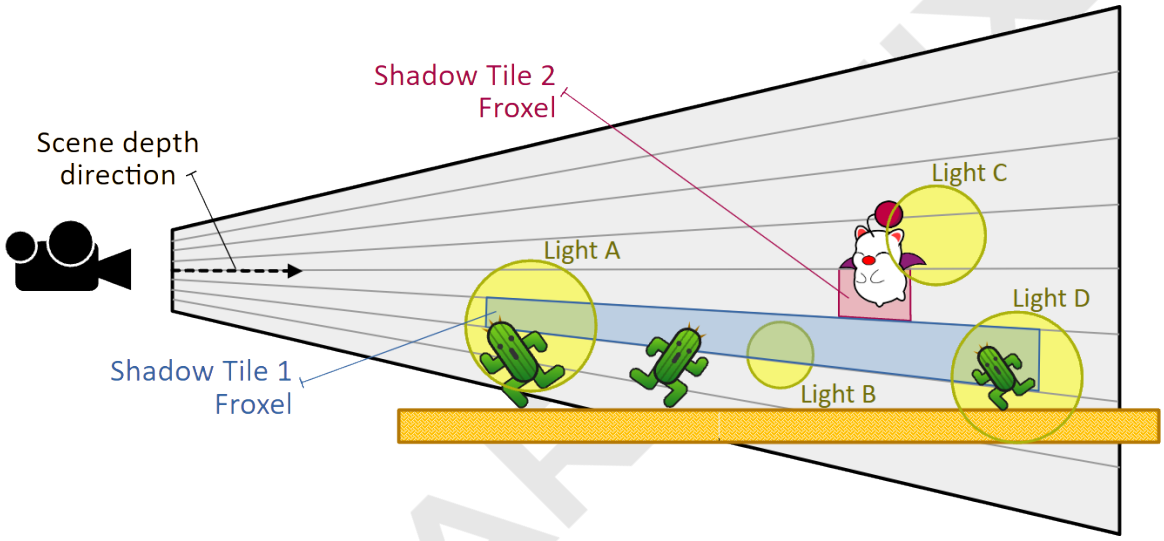


Figure 6: **Accuracy error in approximation of lights found in a *Shadow Tile*.** Two *Froxels* are illustrated here, delimited by their *Shadow Tile* bound plus the minimum and maximum pixel depth. In *Shadow Tile* 1, the lights *A* & *D* are the only ones found when using an accurate test. With the quicker approximation, lights *B* & *C* are also detected incorrectly (no influence on pixels of this *Shadow Tile*). Light *B* because it is inside the *Shadow Tile Froxel*. And light *C*, because it is valid in a neighbor and results are shared between 8x8 *Tiles*.

2.3.1 Output: Early Light List Buffer

This buffer contains information on the lights found in each *Shadow Tile*. To minimize memory usage, we store 3 indices per 4 Bytes, giving us 10 usable bits for a maximum index value of 1023 (*see figure 7a for details*). An additional buffer is also written with the number of valid lights in each *Shadow Tile* list.

	Bits		
	0 ... 9	10 ... 19	20 ... 29
0	Light 1 idx	Light 2 idx	Light 3 idx
1	Light 4 idx	Light 5 idx	Light 6 idx
...	---	---	--

(a) **Data format for 1 *Shadow Tile* entry in the *Early Light List buffer***. It contains a list of valid light indices found in this *Tile* during the *Find Lights Accurate* phase.

	Bits	
	0 ... 9	10 ... 31
0	Light 1 idx	VisibilitySlotID / Flag
1	Light 2 idx	VisibilitySlotID / Flag
...	---	---

(b) **Data format for 1 *Shadow Tile* entry in the *Final Light List Buffer***. It contains a list of valid light indices found in this *Tile* during the *Generate Shadows* phase by processing lights found in the *Early Light List buffer*. It additionally contains the *Visibility SlotID* used to retrieve the per pixel visibility or a flag marking the entire *Tile* to be in shadow or light.

Figure 7: Entries formats for two types of light lists generated per *Shadow Tile*. Each list's memory is allocated by the *Find Lights Approximate* phase using the *Tile* approximated light count, but the actual number stored can be less since the phases outputting them are more accurate.

```
ByteAddressBuffer    tTileShadowSlotIds
ByteAddressBuffer    tTileGroupLightMask;
RWByteAddressBuffer  rw_TileLightListEarly;
RWByteAddressBuffer  rw_TileLightCountEarly;

// Work Unit outputs 1 Tile (8x8 pixels), each thread evaluates 1 pixel
[numthreads(SHADOWTILE_SIZE, SHADOWTILE_SIZE, 1)]
void CS_Phase3_FindLightsAccurate( uint inThreadIdx   : SV_GroupIndex,
                                   uint2 inTileCoord  : SV_GroupID,
                                   uint2 inPixelCoord : SV_DispatchThreadID)
{
  const uint tileIdx      = GetTileIndex(inTileCoord);
  const uint shadowSlotID = tTileShadowSlotIds.Load(tileIdx*4);
  const float depth       = GetDepth(inPixelCoord);
  const float3 worldPos   = GetWorldPosFromScreen(inPixelCoord, depth);
  const bool isValidPixel = !IsFar(depth) &&
                            all(inPixelCoord < cCommon.m_Resolution);
  uint lightMaskAdr       = GetLightMaskGroupAdress(tileIdx);
  uint lightGrpMask       = tTileGroupLightMask.Load(lightMaskAdr);
  uint lightCount         = 0; // Number of valid light found in Tile
  uint lightIdxPacked     = 0; // Pack multiple light indices per uint
  //-------------------------------------------------------------------
  // Process each valid group of 32 lights
  while (lightGrpMask != 0) {
    uint lightGrpBit = firstbitlow(lightGrpMask);
    lightMaskAdr     = GetLightMaskEntryAdress(tileIdx, lightGrpBit);
    uint lightMask   = tTileGroupLightMask.Load(lightMaskAdr);
    lightGrpMask    ^= (1<<firstGroupBit); // Remove processed group
    //-------------------------------------------------------------------
    // Process each valid light in current group
    while (lightMask != 0) {
      uint lightBit       = firstbitlow(lightMask);
      uint lightIdx       = (lightGrpBit * 32) + lightBit;
      LightInfo lightInfo = GetLightInfo(lightIdx);
      lightMask          ^= 1u << lightBit; // Remove processed light
       // Check if pixel is in range of the light
      bool bValidLight    = isValidPixel &&
                            LightPixelTests(lightInfo, worldPos);
      // Add light indice to our list
      if( WaveActiveAnyTrue(bValidLight) && WaveIsFirstLane() ){
        lightIdxPacked = lightIdxPacked | (lightIdx << 10*(lightCount%3));
        lightCount++;
        if( (lightCount % 3) == 0 ){
          uint outputAdr = GetListAddress(shadowSlotID, lightCount);
          rw_TileLightListEarly.Store(outputAdr, lightIdxPacked);
          lightIdxPacked = 0;
        }
      }
    }
  }
  //-------------------------------------------------------------------
  // Store Tile's Light Count in early Light List
  if( WaveIsFirstLane() ){
    rw_TileLightCountEarly.Store(tileIdx*4, lightCount);
    if ((lightCount % 3) != 0) { // output pending packed indices
      uint outputAdr = GetListAddress(shadowSlotID, lightCount);
      rw_TileLightListEarly.Store(outputAdr, lightIdxPacked);
    }
  }
}
```

Listing 3: 'Find Lights Accurate' compute shader pseudo code

2.4 Phase: Generate Deferred Shadows

The 4^{th} phase computes the per pixel visibility of each light with code imported from the original deferred lighting shader. Using the list of lights from the *Shadow Tile* entry in the *Early Light List buffer*, we iterate them to generate shadows using *Percentage Closer Soft Shadows*, but it can be substituted by any other technique.

To reduce the memory usage of our shadow results, we again use a flexible storage approach to avoid allocating one fullscreen buffer for each of the maximum number of lights per *Shadow Tile* supported. This allows us to reduce our buffer size from ~253MB to ~63MB. Instead of having a fixed maximum number of lights, each *Shadow Tile* has as many light results as needed using storage in a shared buffer. We pick a buffer size sufficient to handle the game's worst-case scenario and could be easily modified to adjust dynamically with the number of lights in the level.

We use a dispatch group size of 8x8 to handle one *Shadow Tile* per work unit and with each thread handling a single pixel. When a *Shadow Tile* is entirely obscured, it is ignored and never outputted. When a *Shadow Tile* is entirely in light, we output the light index and a *fully lit* flag. Avoiding light/pixel information storage in these two cases saves a large amount of memory. When a *Shadow Tile* contains partial shadows, we store the per pixel results by assigning it a *Visibility SlotID* computed from a shared global counter with an *atomic operation*. The light visibility ratio uses 8 bits per pixel, so each *Visibility Slot* entry needs 64 Bytes of memory.

2.4.1 Output: *Final Light List Buffer*

This buffer stores per *Shadow Tile* entries and contains the indices of the visible lights that are not entirely in shadow. Along each light index, we store the *fully lit* flag or the *Visibility SlotID* when allocated. 10 bits are used for the light index and 22 bits for the *Visibility SlotID* (*see figure 7b*), enough for ~4 million entries or 32 partial shadow lights per *Shadow Tile* in 4k resolution. In practice, most of the Light/Tile pairs do not need a *Visibility Slot*, so we can support a higher number of lights per *Shadow Tile*, only limited by the size of the *VisibilityBuffer*. An additional buffer is also written with the number of valid lights in each *Shadow Tiles* list.

2.4.2 Output: *VisibilityBuffer*

This buffer contains 8x8 memory blocks used to store the per pixels light visibility of a *Shadow Tile*. This is the largest intermediate buffer needed by the deferred shadow system and can be discarded for the remainder of the frame, once the lighting phase is completed (same is true for the other buffers). Each pixel uses 8 bits to represent the percentage of light visibility, but 4 bits might be sufficient quality depending on the needs of a project.

```
ByteAddressBuffer    tTileShadowSlotIds
ByteAddressBuffer    tTileLightListEarly;
ByteAddressBuffer    tTileLightCountEarly;
RWByteAddressBuffer  rw_TileLightListFinal;
RWByteAddressBuffer  rw_TileLightCountFinal;
RWByteAddressBuffer  rw_TileLightVisibility;

// Work Unit outputs 1 Tile (8x8 pixels), each thread evaluates 1 pixel
[numthreads(SHADOWTILE_SIZE, SHADOWTILE_SIZE, 1)]
void CS_Phase4_GenerateShadows( uint inThreadIdx   : SV_GroupIndex,
                                uint2 inTileCoord  : SV_GroupID,
                                uint2 inPixelCoord : SV_DispatchThreadID)
{
  const uint tileIdx         = GetTileIndex(inTileCoord);
  const uint shadowSlotID    = tTileShadowSlotIds.Load(tileIdx*4);
  const uint earlyLightCount = tTileLightCountEarly.Load(tileIdx*4);
  const float depth          = GetDepth(inPixelCoord);
  const float3 worldPos      = GetWorldPosFromScreen(inPixelCoord, depth);
  const bool isValid         = !IsFar(depth) &&
                               all(inPixelCoord < cCommon.m_Resolution);
  //-----------------------------------------------------------------------
  // Iterate every light previously detected (in Early List light)
  uint finalLightCount  = 0;
  uint entryIdx         = 0;
  while( entryIdx < earlyLightCount ){
    uint lightIdx       = GetNextEarlyLight(shadowSlotID, entryIdx);
    // Standard Shadow generation per pixel happen here
    float viz           = ComputeLightVisibility(lightIdx, worldPos);
    bool isAllLit       = WaveActiveAllTrue(viz > 0.9999 || !isValid);
    bool isAllShadow    = WaveActiveAllTrue(viz < 0.0001 || !isValid);
    bool hasPenumbra    = !isAllLit && !isAllShadow;
    //-------------------------------------------------------------------
    // Only write out light if not entirely in shadow
    if( !isAllShadow ){
      uint vizSlotID = SLOTID_ALL_LIT;
       // Per Pixel visibility generation when partial shadow detected
      if( hasPenumbra ){
        vizSlotID = AllocateVisibilitySlotID(); // 1 Slot/tile
        StoreVisibility(vizSlotID, inThreadIdx, visibility);
      }
      // Output packed light index and VisibilitySlotID/Flag
      uint packedValue = PackLightIndexFinal(lightIdx, vizSlotID);
      uint outputAdr   = GetListFinalAddress(shadowSlotID, finalLightCount);
      rw_TileLightListFinal.Store(outputAdr, packedValue);
      finalLightCount++;
    }
    entryIdx++;
  }
  //-----------------------------------------------------------------------
  // Store Tile's Light Count in Final Light List
  if( WaveIsFirstLane() ){
    rw_TileLightCountFinal.Store(tileIdx*4, finalLightCount);
  }
}
```

Listing 4: 'Generate Deferred Shadows' compute shader pseudo code

2.5 Phase: High Quality Shadows

The final 5^{th} phase processes higher quality shadows of important characters. They are withheld from the *Standard Shadowmaps* and drawn in dedicated *Closeup Shadowmaps* instead. Using a custom *field of view* constrained to the character bound rather than the light leads to a smaller solid angle per pixel and improves resolution. To avoid the high memory overhead of allocating one *Closeup Shadowmap* per light/character pair, we re-use the same one after completing a full update loop (*see figure 8*) and composite the character shadow over the existing *VisibilityBuffer*. Given the scope of this topic, I present an in-depth explanation in section 3.

2.5.1 Output: Updated *Final Light List Buffer*

When a *Shadow Tile* goes from fully lit to partially lit, a *Visibility Slot* needs to be allocated and the *Visibility SlotID* saved. Conversely, a *Shadow Tile* that becomes fully unlit needs to be known by replacing the *Visibility SlotID* with a *fully unlit* flag. Both cases are handled by updating the *Shadow Tile* entry in the *Final Light List Buffer*.

2.5.2 Output: Updated *VisibilityBuffer*

Light visibility is computed using the same technique presented in section 2.4.2 but using the *Closeup Shadowmap* for shadow generation. When a light/pixel visibility is reduced, its value in the *VisibilityBuffer* gets updated.

2.5.3 Output: Updated *Standard Shadowmap*

We avoid increasing the draw count of a character with *High-Quality Shadow* by skipping the *Standard Shadowmap* and only rendering them in the *Closeup Shadowmap*. However, forward-lit objects cannot rely on deferred lighting, so the *Standard Shadowmap* needs to have each *Closeup Shadowmap* composited over its content. This makes the character shadow available to the forward renderer but without the higher-quality benefit.

3 High Quality Shadow

Final Fantasy XVI's emphasis on characters using cinematic close-ups on characters during dialogues makes it important to have higher quality shadows without flickering artefacts distracting players from the scene. This can be achieved by using a separate *Closeup Shadowmap* for each character / light pairing. We want support for an unlimited number of pairings without a high memory overhead proportional to their number. Building on our *Tiled Deferred Shadow* implementation, we have a *High-Quality Shadow* system incrementally processing each character/light shadow with a single reused *Closeup Shadowmap*. This phase contains 5 sub-steps described in the following sections.

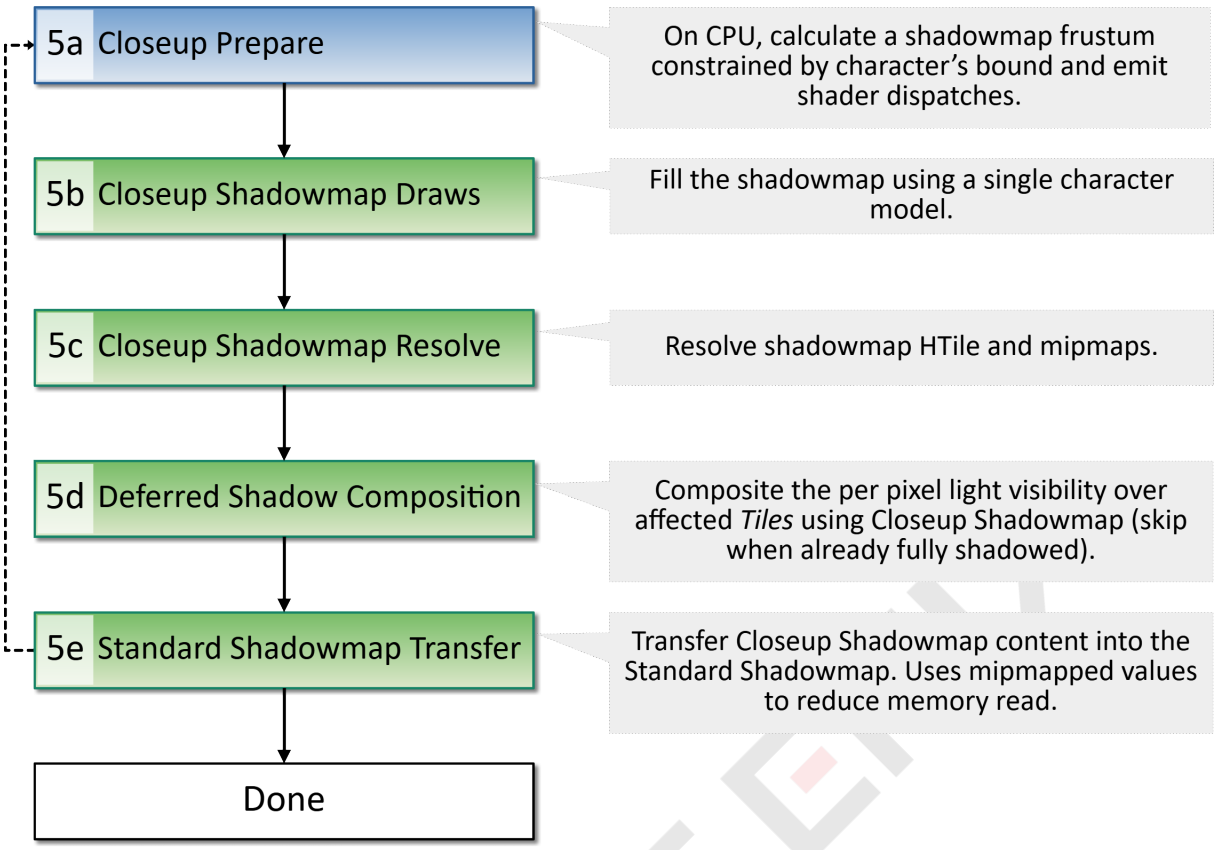
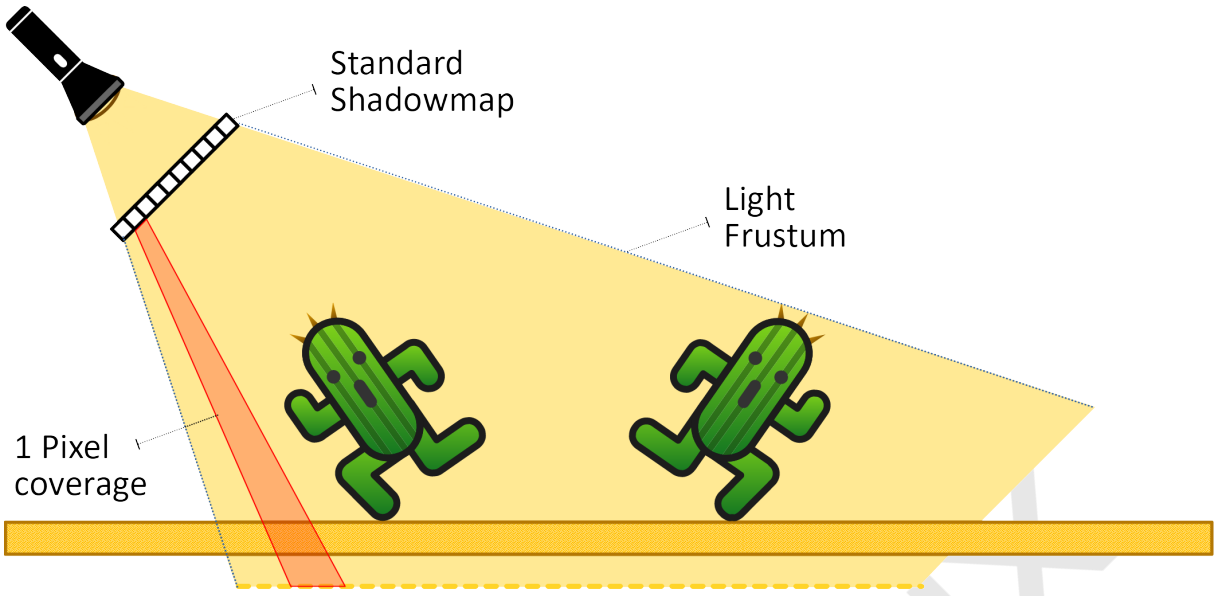


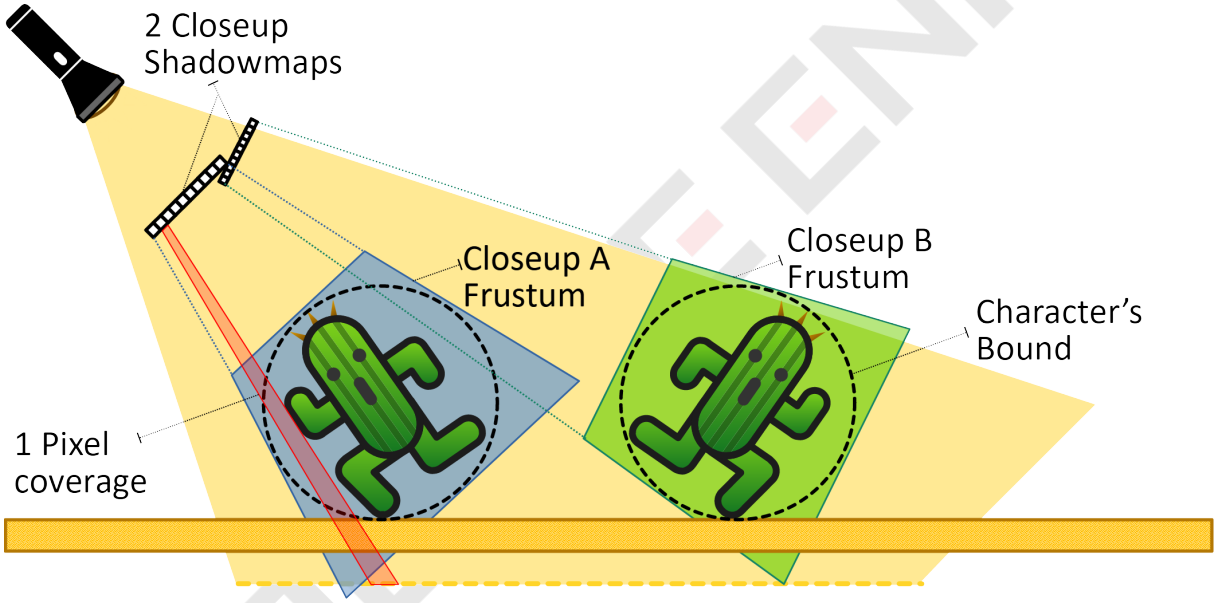
Figure 8: **Closeup rendering flow:** Rendering steps needed to generate shadows for each Closeup (character/light pairs). We iterate over each pair until they all have been handled.

3.1 Step: Prepare

On the CPU side, we calculate a new camera view whose frustum is constrained by the smallest field of view between the character's bound and the light's bound (*see figure 9*). Since we are using a separate *Shadowmap*, its resolution can be higher than the light's *Standard Shadowmap*, leading to even more quality by decreasing the solid angle per pixel further. Our *Closeup Shadowmap* is allocated with 2048x2048 pixels, but each *High-Quality Shadow* adjusts down the resolution used based on the light's estimated screen coverage. Finally, we send the commands to clear the *Closeup Shadowmap* on the GPU at the start of each *High-Quality Shadow*.



(a) Without any Closeup.



(b) With two Closeups.

Figure 9: With *High-Quality Shadows* assigned to light/cactuar pairs, we improve the *Shadowmap* resolution use with a narrower field of view, so pixels handle less solid angle (see red area covered by 1 pixel). The depth precision is also increased by moving the near and far planes closer to the subject.

3.2 Step: Closeup Shadowmap Draws

We avoid drawing the character mesh more than once per light by redirecting its drawcalls from the light's *Standard Shadowmap* to the *High-Quality Shadow*'s *Closeup Shadowmap*. How this is done varies on each rendering engine's implementation.

3.3 Step: Closeup Shadowmap Resolve

Once the *Closeup Shadowmap* draws are completed, we resolve the associated hierarchical depth buffer and generate a mipmapped version of the *Shadowmap*, keeping the nearest distance to the light at each 2x2 pixels.

3.4 Step: Shadows Composition

On the CPU, we build the character's umbra volume by extending rays from the light origin to each bound's positions until the light radius is reached. After projection in screen coordinates, we find the minimum and maximum positions to launch *compute* work units limited to the affected *Shadow Tiles* (*see figure 10*).

In the *compute* shader, each work unit looks for the associated light index in its *Final Light List buffer* entry. When not found or flagged to be entirely in shadow, we stop work here. Otherwise, we evaluate each pixel light visibility and if the entire *Shadow Tile* is fully lit, we stop here. When partial shadow is detected, we ensure that a *Visibility Slot* is allocated, and store the new decreased light visibility using the method presented in section 2.4.2.

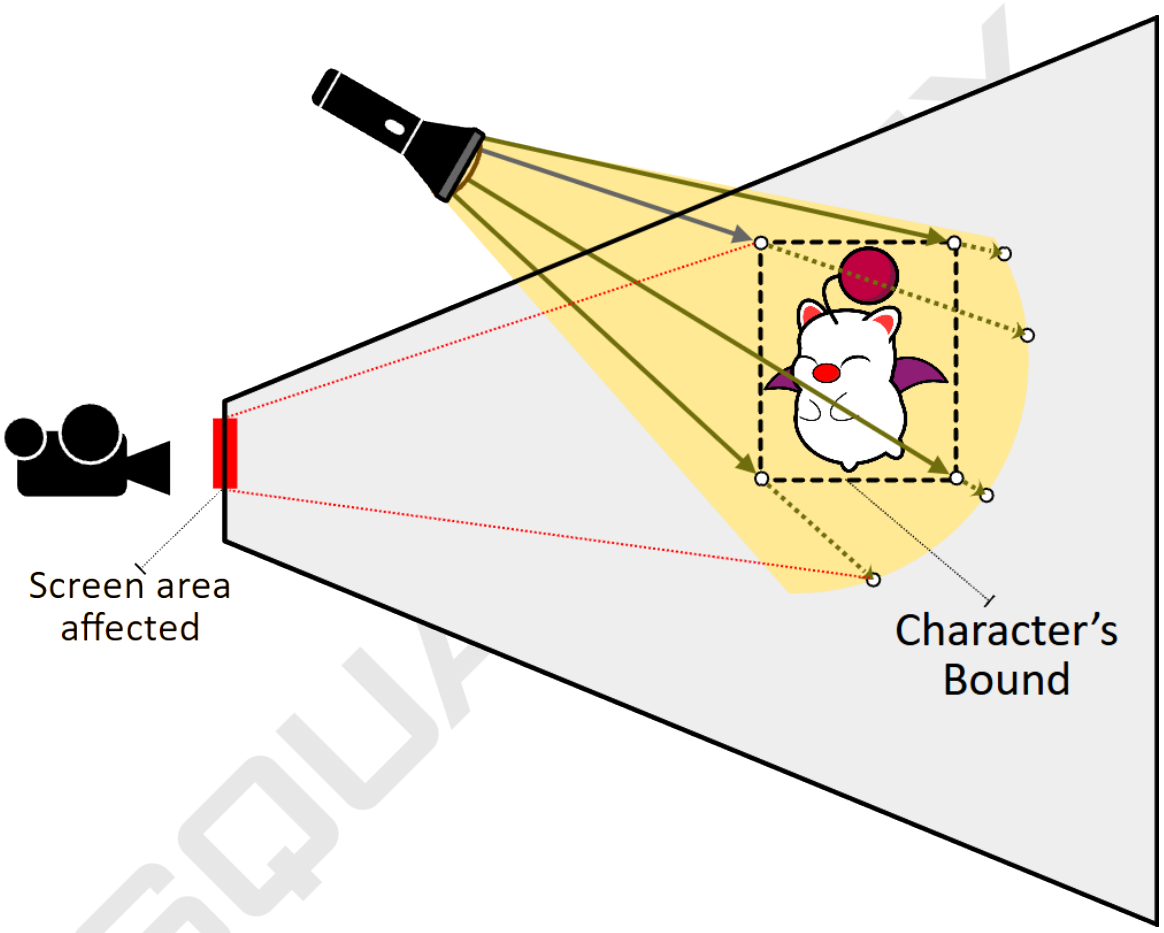


Figure 10: Detection of character umbra volume in screen coordinates.

```
ByteAddressBuffer    tTileShadowSlotIds
ByteAddressBuffer    tTileGroupLightMask;
ByteAddressBuffer    tTileLightCountFinal;
RWByteAddressBuffer  rw_TileLightListFinal;
RWByteAddressBuffer  rw_TileLightVisibility;

// Work Unit outputs 1 Tile (8x8 pixels), each thread evaluates 1 pixel
[numthreads(SHADOWTILE_SIZE, SHADOWTILE_SIZE, 1)]
void CS_Phase5_HighQuality( uint inThreadIdx   : SV_GroupIndex,
                            uint2 inTileCoord  : SV_GroupID,
                            uint2 inPixelCoord : SV_DispatchThreadID
)
{
  const uint tileIdx      = GetTileIndex(inTileCoord);
  const uint tileGrpIdx   = GetTileGroupIndex(inTileCoord);
  const uint shadowSlotID = tTileShadowSlotIds.Load(tileIdx*4);
  const uint lightCount   = tTileLightCountFinal.Load(tileIdx*4);
  const float depth       = GetDepth(inPixelCoord);
  const float3 worldPos   = GetWorldPosFromScreen(inPixelCoord, depth);
  const uint lightIdx     = cCloseupInfo.m_LightIndex;
  bool isValid            = !IsFar(depth) &&
                            all(inPixelCoord < cCommon.m_Resolution);
  //-------------------------------------------------------------------------
  // Stop work when light not detected in TileGroup's LightMask
  if( lightcount > 0 && TestTileGroupLightMask(tileGrpIdx, lightIdx) )
    return;
  // Stop Tile work when pixel is not Closeup
  isValid  &= cCloseupInfo.IsInside(worldPos) &&
              cCloseupInfo.IsValidShadowmapUV(worldPos);
  if( WaveActiveAllTrue(!isValid) )
    return;

  //-------------------------------------------------------------------------
  // Leverage 64 threads to quickly find entry list with light index
  uint vizSlotID = SLOTID_ALL_UNLIT;
  uint entryIdx  = 0xFFFFFFFF;
  for(uint i=inThreadIdx; i<lightcount; i += GROUP_THREAD_COUNT){
    uint2 lightIdx_VizSlot = GetLightListFinalEntry(shadowSlotID, i);
    if( lightIdx_VizSlot.x == lightIdx ){
      entryIdx  = i;
      vizSlotID = lightIdx_VizSlot.y;
     }
  }

  // Stop work when light is not found in this Tile
  entryIdx  = WaveActiveMin(entryIdx);
  vizSlotID = WaveActiveMin(vizSlotID);
  if( entryIdx == 0xFFFFFFFF )
    return;

  //-------------------------------------------------------------------------
  // Update each Tile's pixels light visibility
  float viz = isValid ? ComputeLightVisibility(lightIdx, worldPos) : 0;
  UpdateVisibility(entryIdx, shadowSlotID, vizSlotID, isValid, viz);
}
```

Listing 5: 'High Quality Shadows' compute shader pseudo code (1/2)

```
// Update light visibility in each pixels in the Tile
void UpdateVisibility( in uint  inEntryIdx,  in uint inShadowSlotID,
                       in uint  inVizSlotID, in bool inIsValid,
                       in float inViz)
{
  const bool isAllLit    = WaveActiveAllTrue(inViz > 0.9999 || !inIsValid);
  const bool isAllShadow = WaveActiveAllTrue(inViz < 0.0001 || !inIsValid);
  const bool hasPenumbra = !isAllLit && !isAllShadow;
  const uint lightIdx    = cCloseupInfo.m_LightIndex;

  // If Closeup is not fully lit, must evaluate new results
  if( !isAllLit ) {
    //-------------------------------------------------------------------
    // Tile not 100% shadows, must update content
    if(hasPenumbra ) {
      //-------------------------------------------------------------------
      // Makes sure a VisibilitySlot is allocated
      float oldViz = 1;
      if( inVizSlotID == SLOTID_ALL_LIT ){
        inVizSlotID      = AllocateVisibilitySlotID();
        uint packedValue = PackLightIndexFinal(lightIdx, inVizSlotID);
        uint outputAdr   = GetListFinalAddress(inShadowSlotID, entryIdx);
        rw_TileLightListFinal.Store(outputAdr, packedValue);
      }
      else{
        oldViz = LoadVisibilitySlot(inThreadIdx, inVizSlotID);
      }
      //-------------------------------------------------------------------
      // Update Tile's pixel visibility if any changed
      if( WaveActiveAnyTrue(viz < oldViz && inIsValid) ){
        StoreVisibility(inVizSlotID, inThreadIdx, min(viz, oldViz));
      }
    }
    //-------------------------------------------------------------------
    // Tile 100% in Shadow, mark the Light as unlit
    else{
      uint packedValue = PackLightIndexFinal(lightIdx, SLOTID_ALL_UNLIT);
      uint outputAdr   = GetListFinalAddress(shadowSlotID, entryIdx);
      rw_TileLightListFinal.Store(outputAdr, packedValue);
    }
  }
}
```

Listing 6: 'High Quality Shadows' compute shader pseudo code (2/2)

3.5 Step: Standard Shadowmap Transfer

Each *High-Quality Shadow* is processed separately reusing the shared *Closeup Shadowmap* render target. This has the benefit of the computational cost being the only limiting factor since the memory overhead is fixed. However, the *Shadowmap*'s content is discarded between each character/light pair, making it unavailable to the forward renderer. Since characters with *High-Quality Shadows* are not drawn in them, we transfer each *Closeup Shadowmap* result back into the associated *Standard Shadowmap*.

This is achieved by projecting the *Closeup Shadowmap*'s near plane quad into the light *Standard Shadowmap* coordinates. One *Standard Shadowmap* pixel contains many *Closeup Shadowmap* pixels and reading each one would be prohibitively expensive (*see figure 11*). This high bandwidth problem is solved by using a mipmapped version of the *Closeup Shadowmap* (*see section 3.3*), with each pixel selecting the mipmap level that needs two samples to cover the smallest XY dimension of the area. We then iterate reading mipmapped *Closeup Shadowmap* values until the entire area has been covered. For example, a pixel projecting to a 15 x 37 pixels area in the *Closeup Shadowmap*, we select mipmap level 3.

$$MipLevel = Ceiling(\log_2(min(width, height)/2))$$

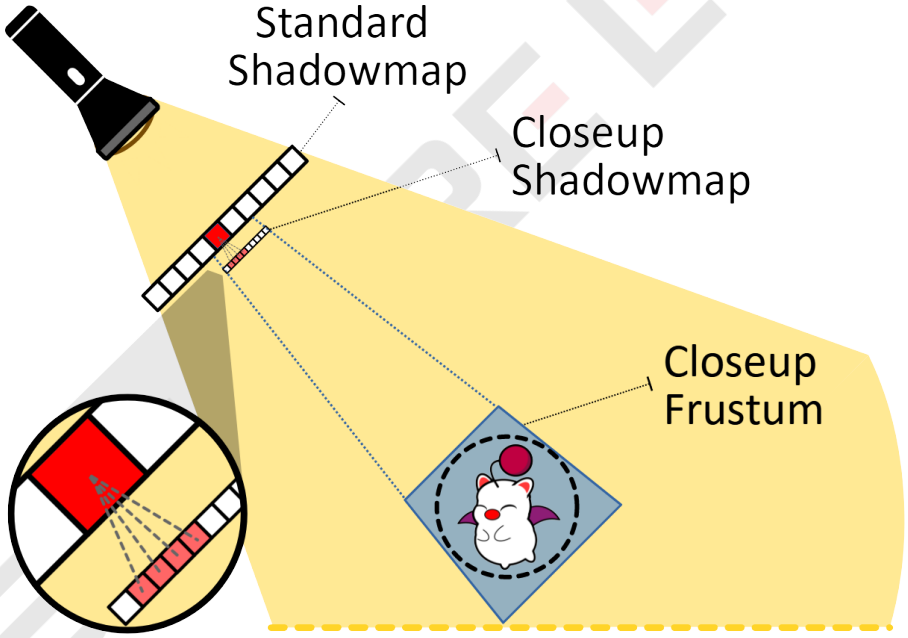


Figure 11: Transfer of a *Closeup Shadowmap* into the *Standard Shadowmap* of a light. Of particular interest is how many pixels are projected into one, so mipmap should be relied on to avoid a high memory bandwidth cost.

3.6 Results

Table 1 demonstrates that there are performance gains achieved by using *Tiled Deferred Shadow* with figure 13 showing the cost distribution of our multi-phase approach. It also shows that *High-Quality Shadows* comes at a cost with figure 14 showing the distribution over each step. This can be managed by modulating the number of allowed character/light pairs. Some venues left to explore for further reductions are resolving multiple *High-Quality Shadows* simultaneously and issuing *Standard Shadowmap* drawcalls normally instead of transferring results to it *(as described in section 3.5)*.

Figure 12: Scene used for performance measurement and averaging 3 shadow lights per tile.

	Shadow	Closeup	Lighting	Total	Δ	
No Deferred Shadows (before)			5.73	5.73		
With Deferred Shadows	2.25		1.69	3.94	-1.79	-31%
Deferred Shadows + 8 Closeups	2.02	2.5	1.74	6.29	+0.56	+10%

Table 1: Deferred Shadows timing comparison on console *(in ms)*. (*Note: Closeups timing includes rendering characters to* Closeup Shadowmaps *and resolving them*).

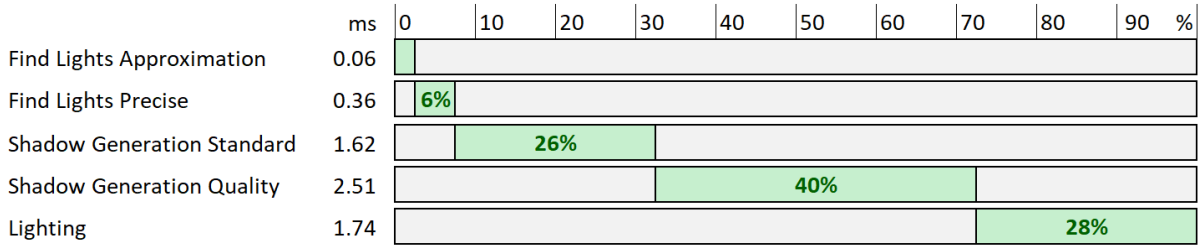


Figure 13: Cost distribution of the *Tiled Deferred Shadow* phases.

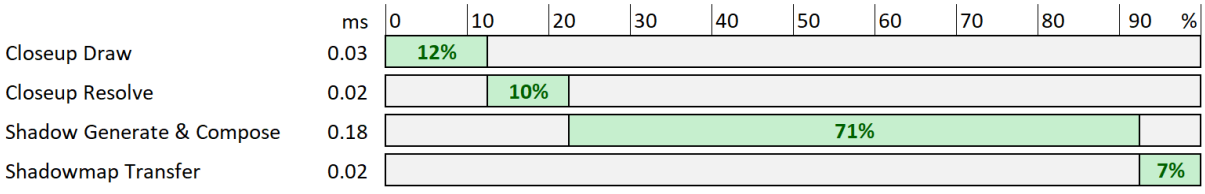


Figure 14: Cost distribution of the *High-Quality Shadow* steps averaged over 8 items.

(a)

(b)

Figure 15: Results of a character's shadow without *(15a)* and with *(15b) High-Quality Shadow*.

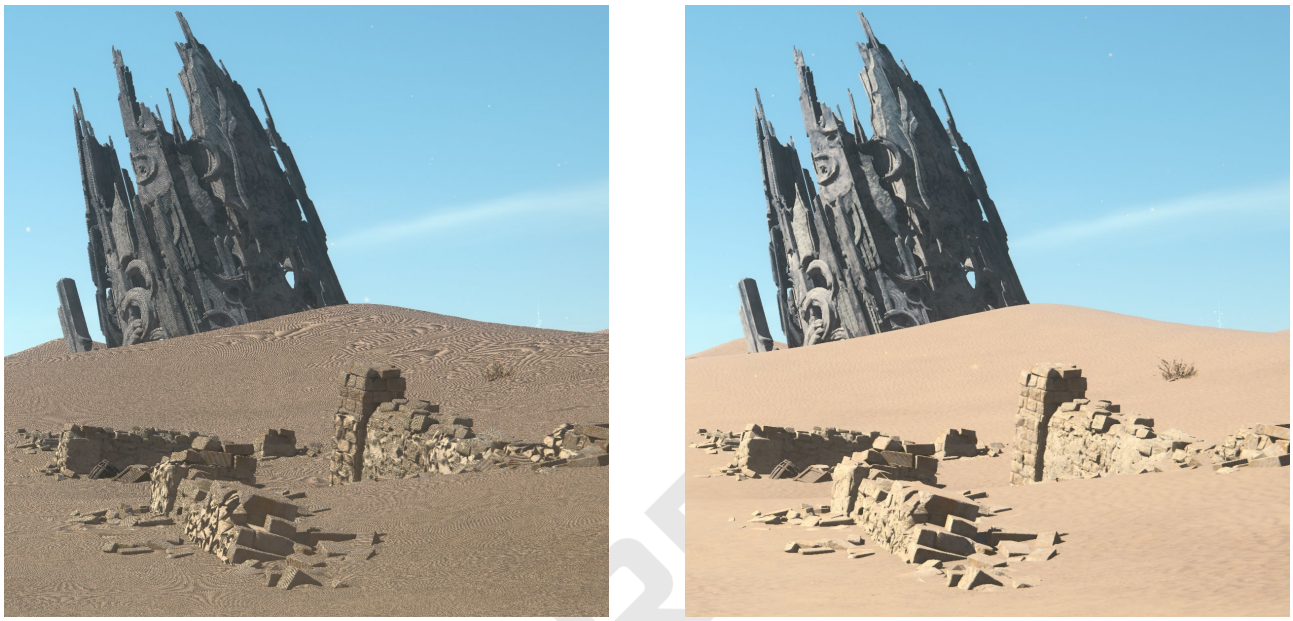
(a)

(b)

Figure 16: Visualization of light's visibility without *(16a)* and with *(16b) High-Quality Shadow*. *Note*: Since hair relies on *Forward Lighting* it cannot use *Tiled Deferred Shadow* and thus cannot receive *High-Quality Shadow*, making them invisible in the visualization.

4 Light leak reduction

The introduction of the *Shadowmap* technique transformed our approach to real time shadows generation and is now used in most games but comes with a few issues *(see 'common techniques to improve shadow depth maps' [5])*. The limited depth-value precision and discretisation of the shadow view when stored in a texture introduces some self-shadow issues (commonly referenced as shadow acne) where a surface erroneously casts some shadows over itself (*see figure 17*). This needs to be mitigated and we introduce a novel approach named *Oriented Depth Bias* to replace the traditional *Hardware Depth Bias*.



(a) Sand casting shadow over itself.

(b) Scene with enough depth bias added.

Figure 17: Shadow acne issues.

(a) A flying chocobo!?

(b) Shadows with proper depth bias amount.

Figure 18: Sample of *Peter Panning* shadow error.

4.1 Hardware Depth Bias

To reduce the shadow acne, GPU allows offsetting the depth written out to the *Shadowmap* render target using *Depth Bias* and *Slope Bias* settings. However, too high an offset introduces a second visual issue called *Peter Panning* where an object seems to float over the ground when its shadows start too far from itself (*see figure 18*). Handling this requires thicker meshes and bias-value tweaking that is a fine balancing between the two problems. Complicating this further is the non-linear aspect of the *Shadowmap* depth *(See 'Depth Precision Visualized' [3] by Nathan Reed)*, making it impossible to pick an appropriate bias value for all depth distances. *Figure 19* illustrates the issues behind the use of *Hardware Depth Bias*.

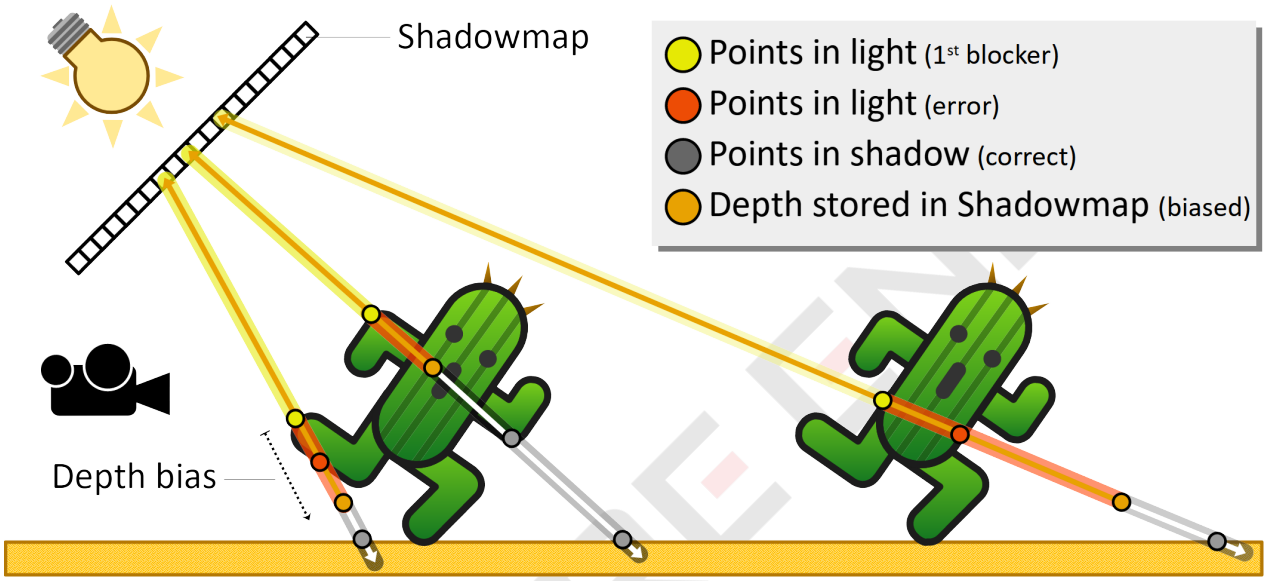


Figure 19: **Hardware Depth Bias** reduce erroneous self shadows but introduce Peter Panning problems where a blocker is thinner than the depth bias (red areas). Despite specifying the bias as a fixed value, we observe that it increases in range the further from the light it is, due to the nonlinear nature of depth projection.

4.2 Oriented Depth Bias

I present a new simple technique solving these issues with little GPU overhead. *Shadowmaps* are first generated without *hardware depth bias* to remain as close as possible to the real surface[3]. We replace it with an *orientation aware bias* added to the tested depth before comparing it with the associated *Shadowmap* value. When evaluating if a point is in shadow, we move it toward or away from the light based on its face orientation. When facing the light, we want to reduce shadow acne and move toward it, decreasing the odds of obstruction. When facing away from the light, we want to reduce Peter Panning and move away from it, increasing the odds of self-shadow. *Figure 20* and *listing 7* illustrate the logic of these two possibilities. This solution lowered our depth bias used to 2 mm and produces reliable shadows on double-sided polygons without thickness.

[3]However, we still rely on a *Slope Bias* value of 1 to handle grazing angle pixels.

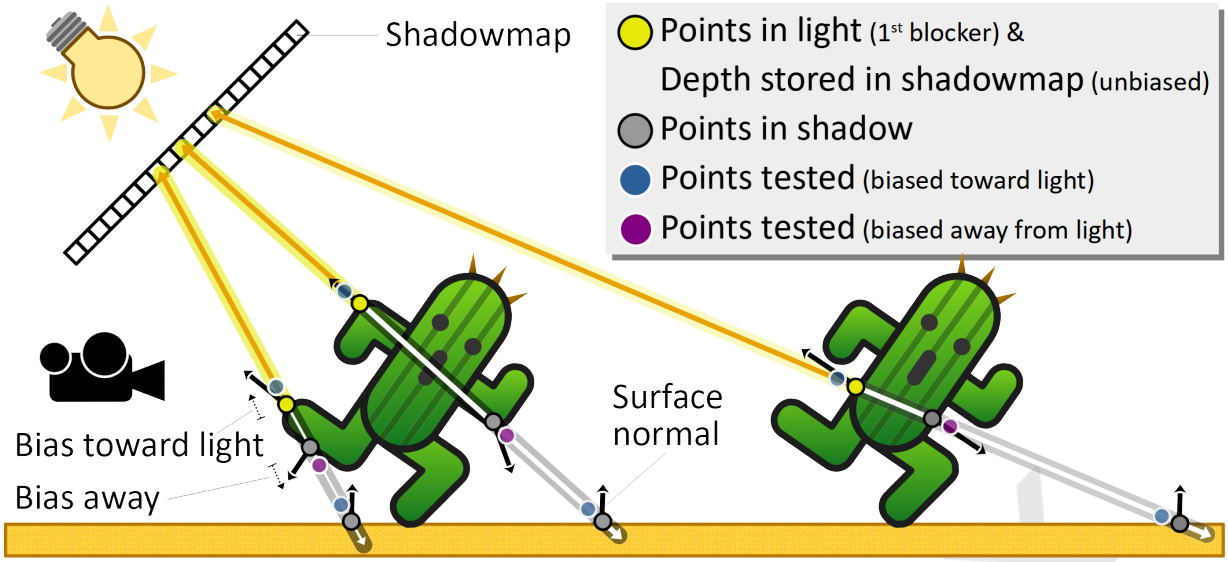


Figure 20: With **Oriented Depth Bias**, shadow acne and *Peter Panning* are eliminated by using an orientation-aware depth bias in linear space. Shadowmaps are generated by storing the first-encountered *unbiased depth* (yellow dots) from the light viewpoint. Next, each shaded point (grey and yellow dots) is biased with a small linear depth value and tested against its associated *Shadowmap* value. This bias moves the point toward the light when its normal is facing it, and away from it otherwise.

We use a *Subsurface Scattering (SSS)* technique to simulate light entering a mesh from a surface location, bouncing inside of it a few times, and exiting at a different location. When looking toward an object in front of a light source, it lets out a diffuse coloration based on its interior tint. It can be observed on thin fleshy parts, tree leaves, some plastic, and so on. SSS requires special handling to remove self shadow that hinders the effect, and this can be achieved by omitting the front face of meshes with SSS when drawing them in the *Shadowmap*. The *Oriented Depth Bias* technique works too well in this case, creating shadows despite precautions. We remedy this during shadow generation by tweaking the technique's behaviour. When handling a pixel with SSS enabled, we change the algorithm to always move toward the light, removing the possibility of self shadows when combined with *Shadowmap* single-face drawing. In the case of wanting to preserve front-face drawing into *Shadowmap*[4] we modify the logic to using a higher depth bias amount when moving toward the light.

[4]This could be to preserve mesh thickness when using Percentage Closer Filtering for example.

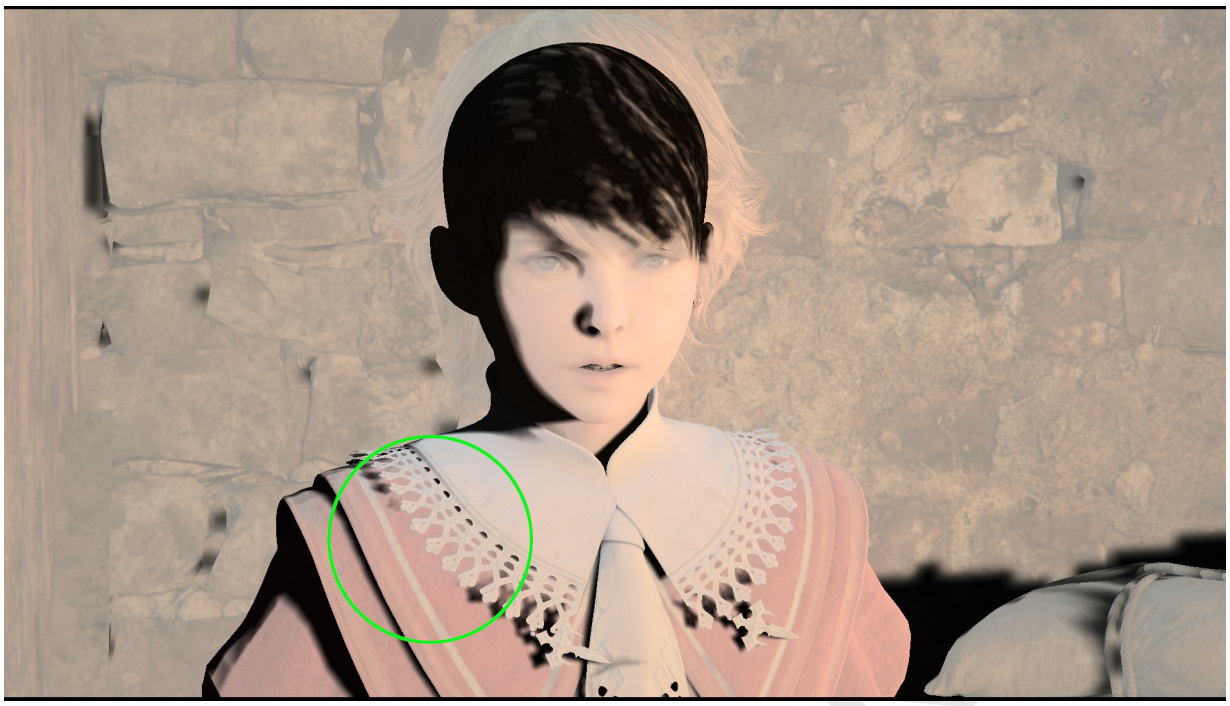
```
float GetOrientedBias( in float3 faceNormal,
                       in float3 lightDirection,
                       in bool isSSS)
{
  static float kOrientedBias = 0.2; // cm (could be a parameter)
  float isFacingLight        = dot(faceNormal, lightDirection) > 0;
  bool moveTowardLight       = isSSS || isFacingLight;
  return moveTowardLight     ? -kOrientedBias : kOrientedBias;
}

float3 GetLightDirection( in LightInfo lightInfo,
                          in float3 worldPos)
{
  return lightInfo.IsDirectional() ? lightInfo.LightDirection
                                   : worldPos - lightInfo.WorldPos;
}

float ComputeShadows( in LightInfo lightInfo,
                      in float3 worldPos,
                      in float3 faceNormal,
                      in bool isSSS )
{
  float3 lightDir       = GetLightDirection(lightInfo, worldPos);
  float3 shadowmapCoord = WorldToShadowmapCoord(lightInfo, worldPos);
  float linearDepth     = DepthToViewZ(lightInfo, shadowmapCoord.z);
  float linearBias      = GetOrientedBias(faceNormal, lightDir, isSSS);
  shadowmapCoord.z      = ViewZToDepth(lightInfo, linearDepth+linearBias);
  return PerformShadowmapTest(lightInfo, shadowmapCoord);
}
```

Listing 7: Shader code logic when applying *Oriented Depth Bias* to the shaded pixel

4.3 Results

	Shadows	Δ	
Hardware Depth Bias (before)	3.98		
Oriented Bias	4.17	+0.19	5%
Oriented Bias (extra GBuffer reads)	4.27	+0.32	8%

Table 2: Oriented Depth Bias timing comparison on console (*in ms*). The last row measures time with extra GBuffer accesses for surface normal and SSS detection when not already available to the shader.



(a)

(b)

Figure 21: Visualisation of light's visibility without *(21a)* and with *(21a) Oriented Depth Bias*. Notice the missing shadow under the lace.

5 Visualizers

Introducing multiple steps and indirection to shadow generation creates difficulties in the diagnostic of issues that will invariably arise during implementation. Therefore, having access to a variety of debug visualizers becomes essential, giving detailed insight into each step and preserving shader debugging capability with pixel picking in a GPU frame capture. The visualizers are used both to display the results in real-time over the game and to output them to a temporary render target for later analysis in a GPU frame capture.

Figure 22: Reference scene with light used in the following visualizer previews.

5.1 Shadow

This is the most used visualizer and shows the shadow of a selected light. A first mode displays the previously computed *Tiled Deferred Shadow* and a second mode re-evaluate the shadow while entirely ignoring the light culling process. This last mode is particularly useful in debugging the shadow generation algorithm (like PCSS) of a specific pixel in a GPU capture.

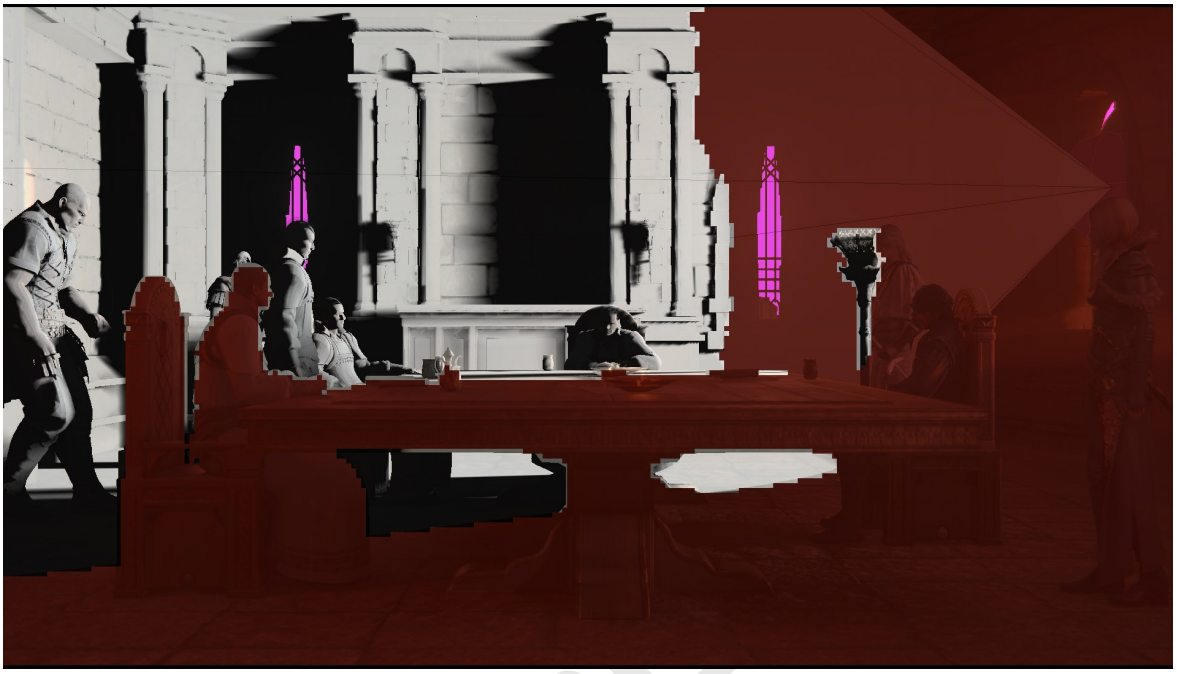


Figure 23: Visualizer: *Tiled Deferred Shadow* results.

5.2 Visibility Slot

This visualizer shows the number of *Visibility Slots* allocated per *Shadow Tile* and is helpful in memory budget analysis.

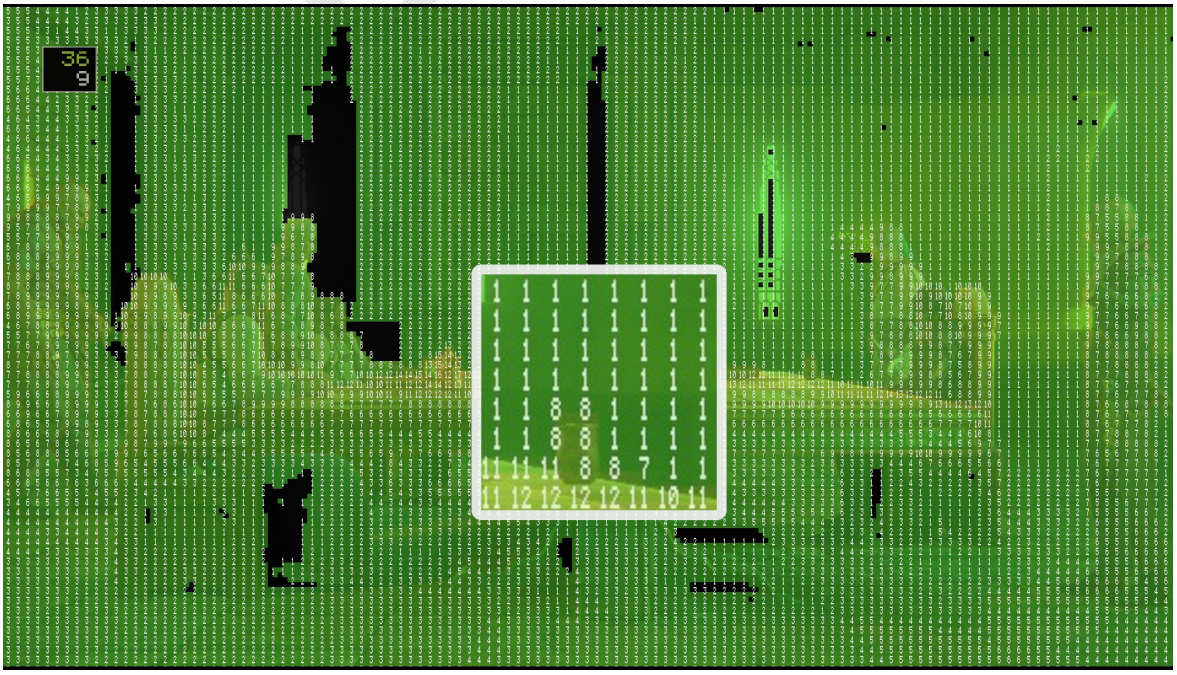
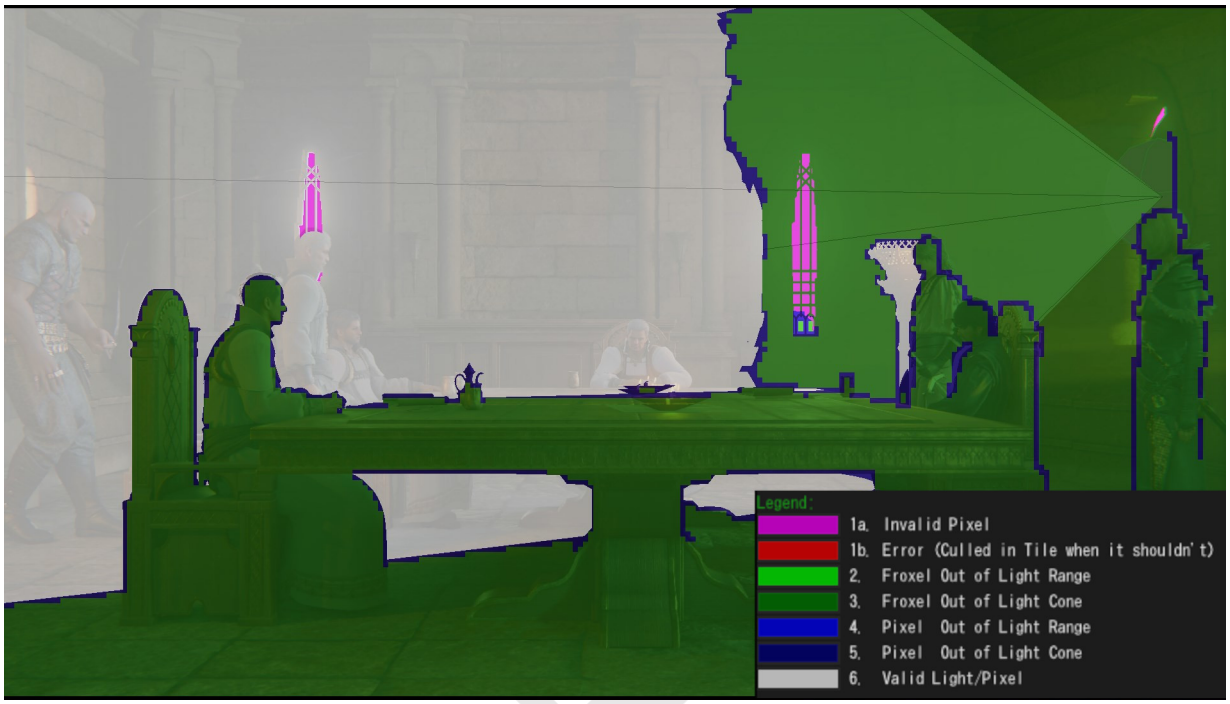


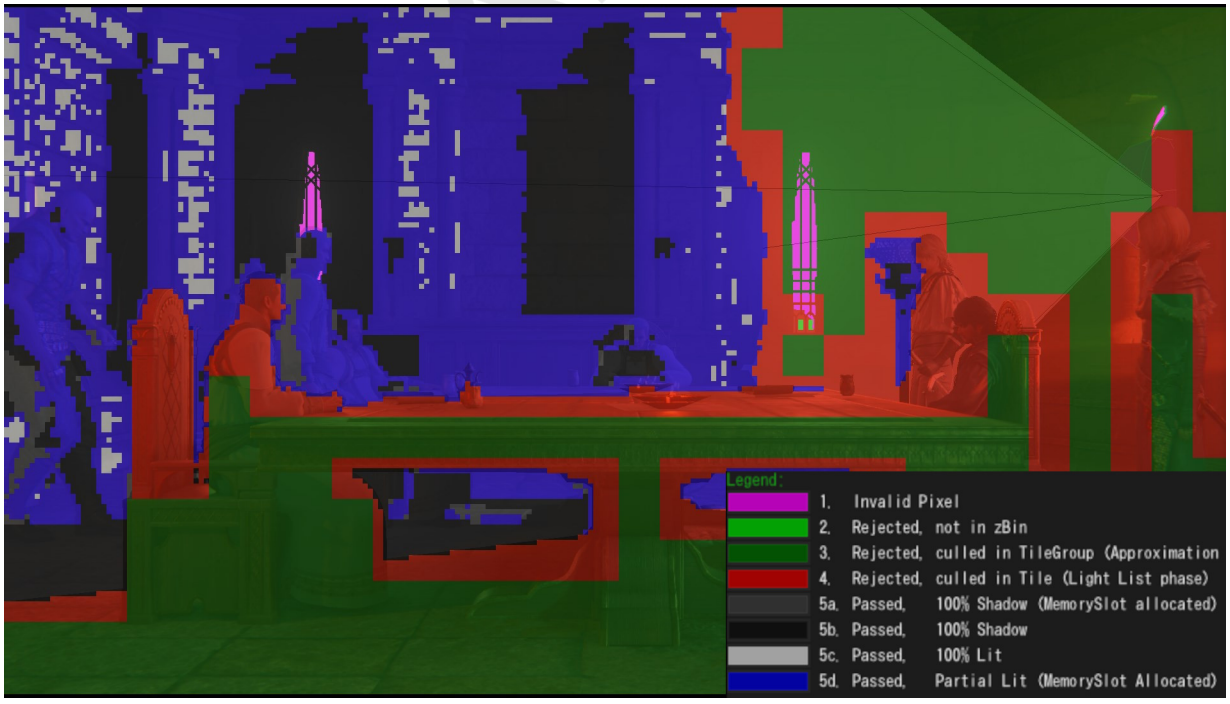
Figure 24: Visualizer: number of *Visibility Slots* allocated per *Shadow Tile*.

5.3 Light Culling

This visualizer previews the result of each light culling step, and which list a light is part of. This is a valuable display helping with diagnostics since the generation of the two light lists *(*Early Light List buffer *and* Final Light List Buffer*)* can encounter many issues. A useful feature is the ability to detect a light erroneously culled by checking if any *Shadow Tile*'s pixels are in range and confirming that the result agrees with the culling tests performed.



(a) Visualize last culling step successfully passed by a light.

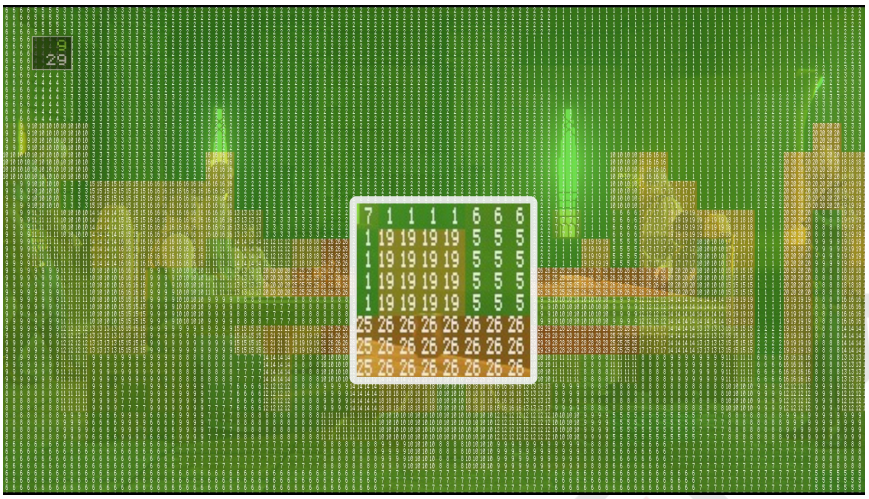


(b) Visualize which light list a light belongs to.

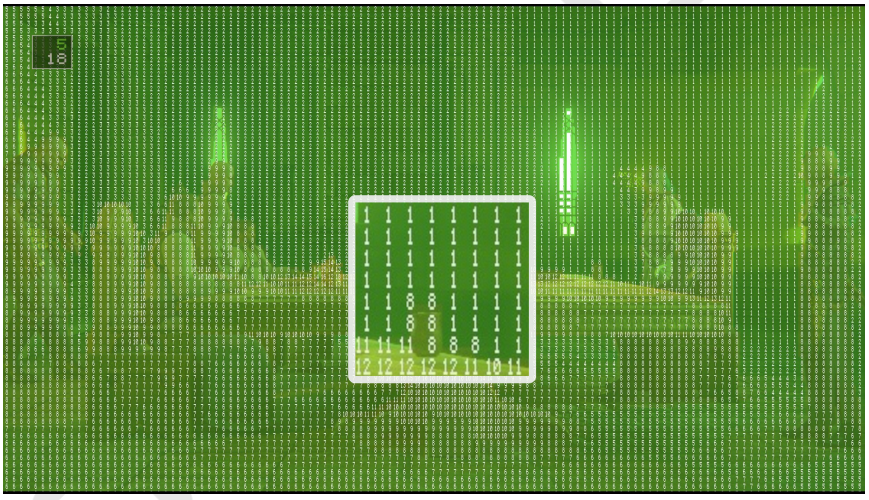
Figure 25: Visualizer: Light culling results.

5.4 Light Count

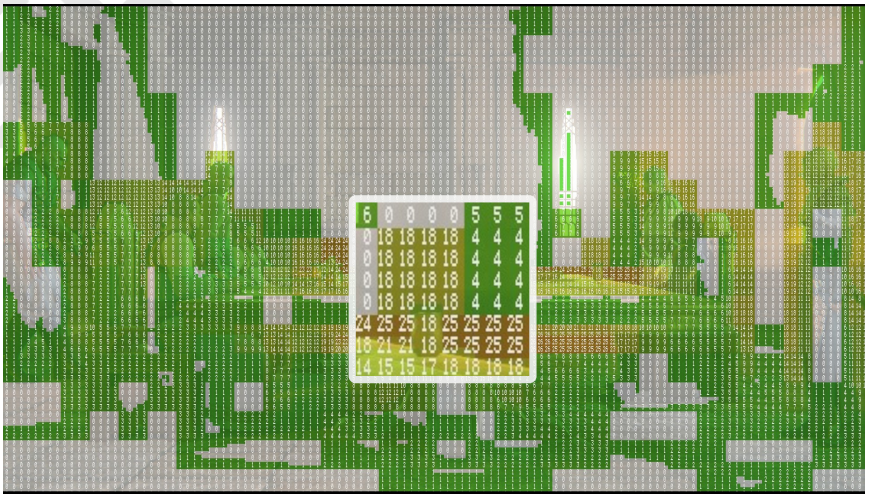
Useful to artists and programmers alike, this visualizer displays the number of lights detected per *Shadow Tile* in various *Tiled Deferred Shadow* phases. Of particular use is the mode displaying the light count difference between the approximation *(section 2.2)* and the accurate *(section 2.3)* phases highlighting where it is weakest.



(a) Approximated number of lights detected.



(b) Number of lights found with more accurate tests.



(c) Difference between the approximated and the real light count.

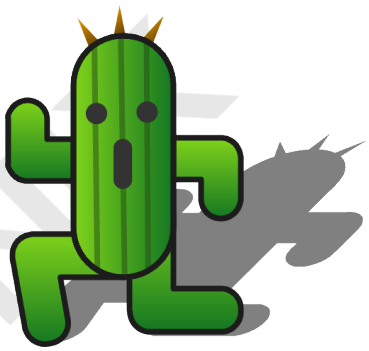
Figure 26: Visualizer: Light count per *Shadow Tile*.

6 Conclusion

We have implemented a *Tiled Deferred Shadow* system resulting in performance gains on standard shadows. It also allows support for characters' *High-Quality Shadow* for an extra cost that we manage by limiting their number based on the available budget. Next, I introduced our *Oriented Depth Bias* technique for simple handling of *Shadowmap* self-shadow issues. Finally, I presented our visualizers helping with *Tiled Deferred Shadow* issues diagnostics.

7 Acknowledgements

The development of any game of this magnitude requires the collaboration of a large number of dedicated individuals. I would like to express my gratitude to Honda Kei for providing a solid foundation of rendering code and knowledge on which I was able to rely during developments. I would also like to thank Hideyuki Kasuga for his management support, the Business Unit 3 team for their trust in allowing me to explore new features, and the graphics members of the Advanced Technology Division for their constructive feedback on this document. Finally, I would like to thank Louis-Philippe Sanschagrin for his valuable artistic feedback and Heather Lee Mills for her document edits.



References

[1] W. Bart. *Cull that cone! Improved cone/spotlight visibility tests for tiled and clustered lighting*. Apr. 2017. URL: https://bartwronski.com/2017/04/13/cull-that-cone (visited on 06/01/2023).

[2] Bo Li. "A Scalable Real-Time Many-Shadowed-Light Rendering System". In: *ACM SIGGRAPH 2019 Talks*. SIGGRAPH '19. Los Angeles, California: Association for Computing Machinery, 2019. ISBN: 9781450363174. DOI: 10.1145/3306307.3328167. URL: https://doi.org/10.1145/3306307.3328167.

[3] R. Nathan. *Depth Precision Visualized*. NVidia. July 2015. URL: https://developer.nvidia.com/content/depth-precision-visualized (visited on 06/02/2023).

[4] Ola Olsson, Markus Billeter, and Ulf Assarsson. "Clustered Deferred and Forward Shading". In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, 2012, 87–96. ISBN: 9783905674415.

[5] W. Steven et al. *Common Techniques to Improve Shadow Depth Maps*. Microsoft. Sept. 2020. URL: https://learn.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps (visited on 06/02/2023).

Glossary

Closeup Shadowmap

Shadowmap used with our *High-Quality Shadow* technique. Operates like a normal *Shadowmap* but with the view restricted to one object of interest rather than the entire light's

field of view for better resolution precision.

Compute Work Unit

GPU speed comes from parallel execution of a series of instructions in lockstep with multiple data. On console we have access to 64 threads running in parallel and are able to quickly exchange data with each other.

Early Light List buffer

GPU buffer storing a list of valid lights per *Shadow Tile*. Lights are added to it when there are pixels in range of the light without performing shadow tests *(see figure 7a for entry format)*.

Final Light List Buffer

GPU buffer storing a list of valid lights per *Shadow Tile*. Lights are added to it when there are pixels in range of the light and excluded if the entire *Tile* is in shadow *(see figure 7b for entry format)*.

Froxel

Name given to *Frustum Voxel* which defines a sub-section of a Frustum using Tile coordinates and a near/far depth position.

Hierarchical Depth Buffer

Special depth buffer used by GPU for acceleration. It has $1/8^{\text{th}}$ of the paired depth buffer resolution and contains the minimum and maximum depth of the associated 8x8 pixels.

High-Quality Shadow

Technique to draw higher quality shadow on specific objects (usually characters). Relies on *Closeup Shadowmaps* focused solely on the object of interest.

Lane Operator

GPU instructions allowing fast data sharing between threads of a *Compute Work Unit*.

Light Group

Group of 32 contiguous lights. Used by *Light Mask Entry* to quickly discard groups without any valid light, reducing processing time and memory read *(see figure 4)*.

Light Mask Entry

Memory block of 132 bytes used to store the validity of 1024 lights and 32 *Light Groups*, using 1 bit per entry *(see figure 4)*.

Light SlotID

Used to calculate the starting location of a Tile's light list inside a memory buffer. Identifier is valid for both the *Early Light List buffer* and the *Final Light List Buffer*.

Oriented Depth Bias

New depth bias technique introduced in section 4.2 that applies an offset to the depth tested using the face orientation.

Percentage Closer Soft Shadows

Shadow technique expanding the use of *Shadowmap* to allow soft shadow mimicking light's penumbra. The shadow becomes more diffuse as the geometry blocking the light increase its distance from it.

Peter Panning

Artefact where object seems detached from its shadow because of a large depth bias applied to the *Shadowmap (see figure 18)*.

Shadow Tile

Tile used for processing *Tiled Deferred Shadow*. Our tiles are sized at 8x8 pixels to account for the 64 threads *Compute Work Unit* size on console's GPU.

Shadowmap

Texture storing the nearest depth as seen from the perspective of a light source. Used in the most common shadow technique in real-time rendering.

Standard Shadowmap

Conventional *Shadowmap* used when generating the shadow of a light.

Tile

2D contiguous pixels grouped together for compute shader processing on the GPU.

Tile Group

2D contiguous *Tiles* grouped together for compute shader processing on the GPU. Used in our approximation of lights per *Tile* search for faster results and in the *Tile Group Light Mask Buffer* for reduced storage space. We use groups of 8x8 *Tiles* to account for the 64 threads *Compute Work Unit* size on console's GPU.

Tile Group Light Mask Buffer

GPU buffer storing the visibility of light in each *Tile Group* after the approximation phase *(see figure 4 for entry format)*.

Tile Light Mask Buffer

GPU buffer storing the visibility of lights in each *Tile*. Unlike our *Shadow Tile* of 8x8 pixels, these *Tiles* are 32x32 pixels to reduce the CPU workload when populating the buffer *(see figure 4 & 5)*.

Tile Light SlotID Buffer

GPU buffer storing the *Light SlotID* of each *Shadow Tile* after assignment in the approximation phase.

Tiled Deferred Lighting

Rendering technique separating lighting calculations from the surface material calculation. This is done by storing the material parameters in intermediate textures called GBuffers and processing the lighting in *Tiles* work units.

Tiled Deferred Shadow

Rendering technique expanding on *Tiled Deferred Lighting* by further removing the shadow calculation from the lighting, then processing it in *Shadow Tiles* work units.

VGPR Pressure

Shaders need to reserve enough registers to accommodate the computational complexity at its heaviest point. By lowering this number, we increase the number of *Compute Work Units* waiting for execution. This helps hide memory access latency by switching to a pending task when waiting for a memory load.

Visibility Slot

Memory block storing the visibility of one light for each pixel of a *Shadow Tile*. Use 8 bits per pixel to show gradient in the light penumbra.

Visibility SlotID

Used to calculate the starting location of a Tile light visibility content inside the *Light Visibility* memory buffer.

VisibilityBuffer

GPU buffer storing the light visibility of each pixel of a *Shadow Tile* for a specific light. When a light is fully lit or unlit, no entry is allocated in this buffer.

zBin

The Frustum is divided in 1024 uniform depth ranges creating zBins that store the first and last light indices intersecting it *(see figure 2a)*.

zBin (ranged)

Contains the first and last light indices intersecting a specific range of contiguous *zBins (see figure 2b)*.

zBin Buffer

GPU buffer storing entries of *zBin* and *zBin (ranged)*.