Deferred Rendering Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Overview

Scope of This Document

This document explains the techniques for implementing deferred rendering on the GPU of the PlayStation®Vita and its performance characteristics.

This document assumes familiarity with real-time rendering.

Structure of this Document

This document is comprised of the following chapters.

- Chapter 2 "Deferred Rendering" describes important considerations regarding the implementation of deferred rendering.
- Chapter 3 "Lighting Techniques" describes important considerations regarding the implementation of lighting techniques used in deferred rendering.
- Chapter 4 "Optimization and Performance" describes the implementation of optimization techniques, as well as performance characteristics.

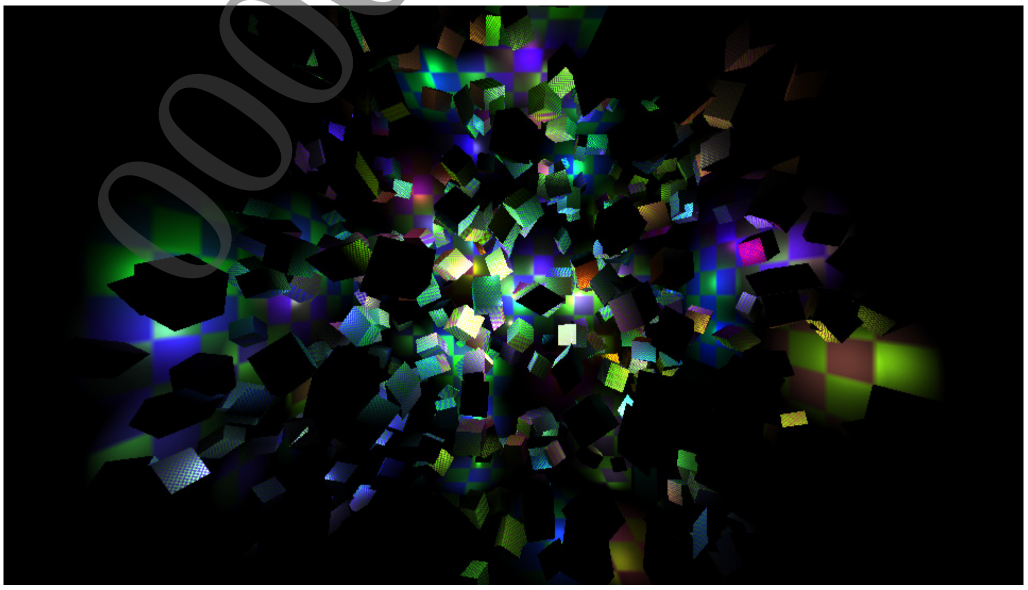
Overview

In games, the traditional method of rendering is to render an object with its associated material shader, computing all lighting processes that affect the object. This technique is called "forward rendering".

In contrast, the idea of "deferred" rendering is to decouple the process of lighting an object from the process of rendering it. This helps to mitigate rendering complexities related to combinations of materials and light sources. It also makes supporting additional light sources more straight-forward.

The scene rendered by the sample program consists of boxes representing opaque scene geometry and point light sources. The point light sources are initialized randomly with different colors and positions. All point light sources have the same radius. The number of light sources and their radius can be changed from the menu of the running sample program. Figure 1 shows an example of a scene rendered by the sample program.

Figure 1 Scene Rendered by the Sample Program



The sample program implements two shading methods:

- Deferred Shading
- Light Prepass

The sample program also implements two methods for lighting with point light sources:

- Light Volumes
- Indexed Lights

Either of these lighting methods can be applied with either of the shading techniques and it is possible to toggle between them while the sample program is running.

Sample Program

All source code implementing the deferred rendering techniques can be found in the following directory:

- sample_code/graphics/tutorial_deferred_rendering

Sample Programs

See the following document for information regarding the PlayStation®Vita shader compiler:

- Shader Compiler User's Guide

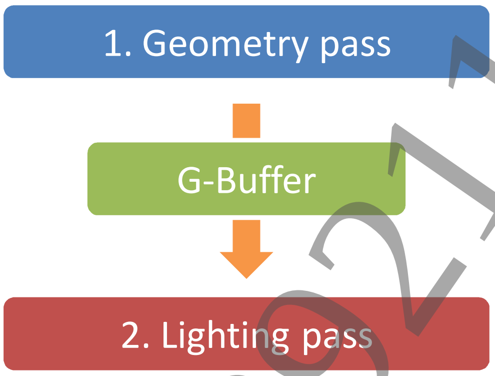
2 Deferred Rendering

This chapter describes two deferred rendering methods: deferred shading and light prepass.

Deferred Shading

Deferred shading method is generally divided into two passes (Figure 2). In the first pass, the geometry is rendered and the information required for the lighting of the final image is stored in a buffer called the G-buffer. This first pass is called the geometry pass. In the second pass, called the lighting pass, the contents of the G-buffer are referenced as a texture to perform the lighting calculation.

Figure 2 Deferred Shading Flow



Building the G-Buffer (Geometry Pass)

In the sample program, a 64-bit color surface is used as the G-buffer. It is initialized as follows, with the surface format set to `SCE_GXM_COLOR_FORMAT_F16F16F16F16_ABGR` and the output register size set to `SCE_GXM_OUTPUT_REGISTER_SIZE_64BIT`.

C++ Code to Create a Color Surface for the G-Buffer

```
gbuffer = CreateColorSurface(
    g_CdramHeap,
    &d.gbufferSurface,
    SCE_GXM_COLOR_FORMAT_F16F16F16F16_ABGR,
    SCE_GXM_COLOR_SURFACE_TILED,
    multisampleMode,
    displayWidth,
    displayHeight,
    SCE_GXM_OUTPUT_REGISTER_SIZE_64BIT);
```

The sample program stores 3xU8 diffuse color, 3xU8 normal, and 1xU8 specular exponent values in the G-buffer, as shown in Figure 3. Since depth value is also necessary in the lighting pass, writing to the memory of the depth buffer is enabled in the geometry pass. The results of rendering these buffers are shown in Figure 4 and Figure 5. The contents of the G-buffer can be displayed during execution of the sample program.

Figure 3 The G-Buffer for Deferred Shading

R8	G8	B8	A8	Format
Diffuse color X	Diffuse color Y	Diffuse color Z	Specular exponent	A8B8G8R8
Normal X	Normal Y	Normal Z		A8B8G8R8
	Depth		Stencil	S8D24

Figure 4 Normal Buffer

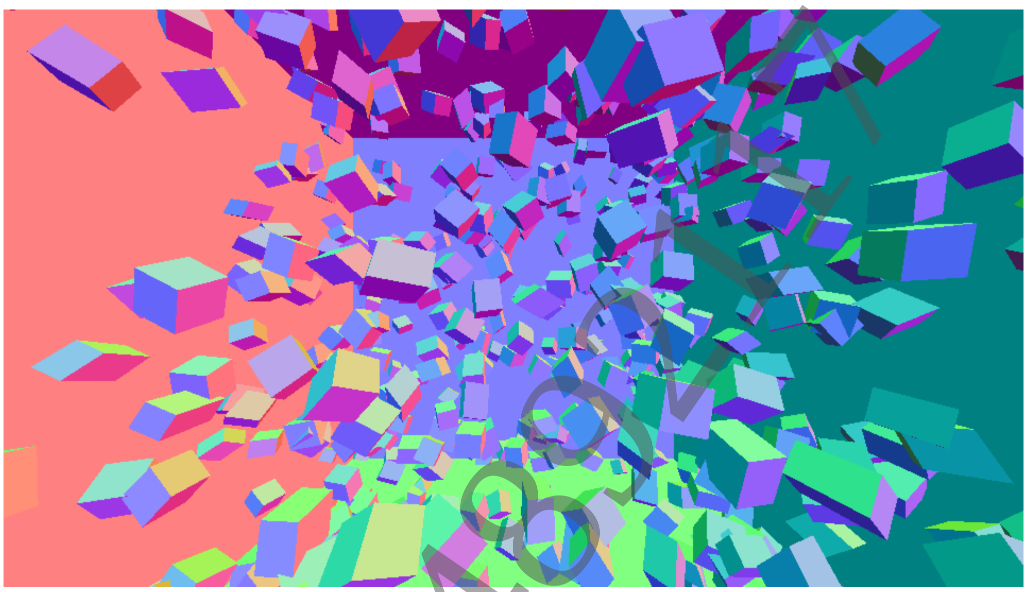
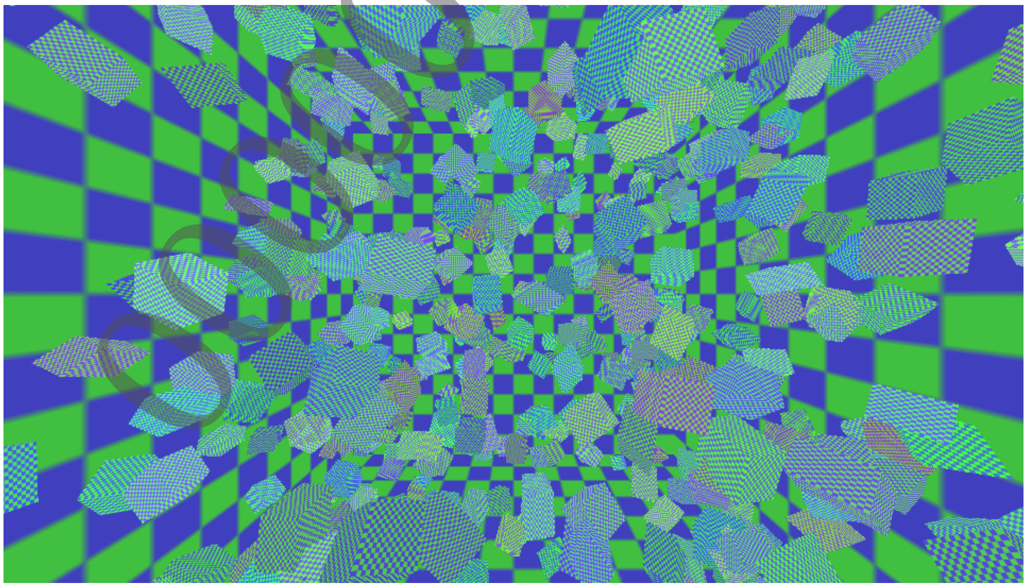


Figure 5 Diffuse Color Buffer



In order to pack this information into the G-buffer, in the fragment shader, the three component of 24-bit diffuse color is encoded into a single component of 32-bit unsigned integer using the `bit_cast` Cg keyword.

Cg Code to Pack the G-Buffer Using bit_cast

```
// Convert [0.0f,1.0f] range to [0,255]
unsigned char4 bytes = diffuseCol.xyzz*255.0;

// Encode the char4 as an unsigned int
unsigned int diffusecolor = bit_cast<unsigned int>(bytes);
```

In the same manner, the sample program encodes the normal as an unsigned integer, and packs these two values into an `unsigned int2`. It then bit-casts this value to `half4`, and writes it to the output register. By using the `__regformat` keyword, an instruction can be made to write the value in the format declared to the Cg compiler and libgxm; so that when `__regformat half4` is written, the fragment shader output will be written as 64-bit `half4` as declared.

Cg Code for Outputting to the G-Buffer by Specifying the type using __regformat

```
__regformat half4 main(
    float2 Texcoord : TEXCOORD0,
    float3 Normal : TEXCOORD1,
    uniform sampler2D Texture0 : TEXUNIT0)
    {
        ...
        gbuffer = bit_cast<half4>(unsigned int2(diffusecolor,normal));
        return gbuffer;
    }
```

Since the maximum size of the output register and color surface which can be handled by the PlayStation®Vita GPU is 64 bits, techniques using a writable uniform buffer should be considered to build a G-buffer that can hold more than this amount of data. A writable uniform buffer is used as an auxiliary surface for storing information needed from the fragment shader. It could also be used, for example, to additionally output the positions of pixels in the lighting space. Since this would also increase the time required to store it in memory, as well as the overhead to reference it in the lighting pass, attention must be paid to performance degradation. For information on handling the positions of pixels in the lighting space, refer to the "Depth Resolve" section of Chapter 4 "Optimization and Performance".

Referencing the G-Buffer (Lighting Pass)

In the lighting pass, the G-buffer is referenced as a texture, and the final lighting is computed. Referencing the G-buffer, which was created in the geometry pass, as a U32U32-format texture, the 3xU8 diffuse color and 3xU8 normal data is unpacked and extracted again. In the same manner as during packing, fast decoding from U32 to 4xU8 can be performed using `bit_cast`.

Cg Code to Unpack the G-Buffer Using bit_cast

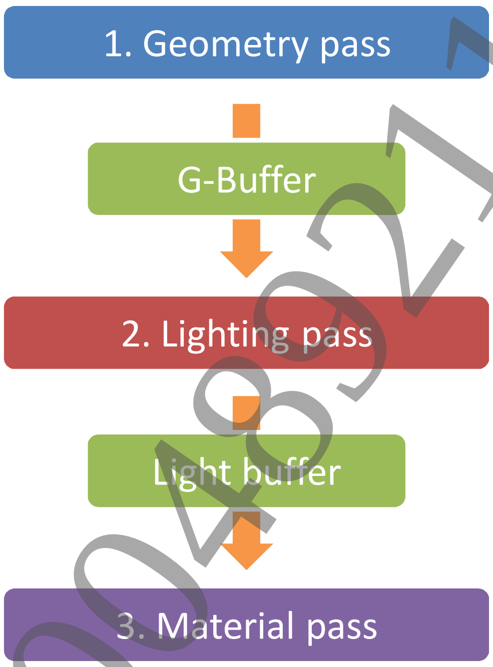
```
// function to extract 4 component float/half from 32 bits uint
real4 decodeFloat4FromInt(unsigned int src)
{
    unsigned char4 bytes = bit_cast<unsigned char4>(src);
    return bytes/255.0;
}
```

For information on unpacking the G-buffer more quickly, refer to the "Split G-Buffer" section of Chapter 4 "Optimization and Performance".

Light Prepass

In contrast to the deferred shading method, the light prepass method renders the geometry twice. As a result, the light prepass method is comprised of three passes (Figure 6). The first pass is the geometry pass, in which the geometry is input as in the deferred shading method, but only the minimum amount of information necessary for computing the lighting is stored in the G-buffer. The second pass is the lighting pass. Lighting is computed based only on light information and by referring to the G-buffer. This data is accumulated into a buffer called the light buffer. The final pass is the material pass, which takes the geometry as input again, and performs the final shading using the data in the light buffer. Since the geometry is input twice in the light prepass method, the cost of vertex processing is higher than the deferred shading method, but since data on multiple materials can be passed to the fragment shader in the material pass, the process is not limited by the G-buffer, and more flexible shading is possible.

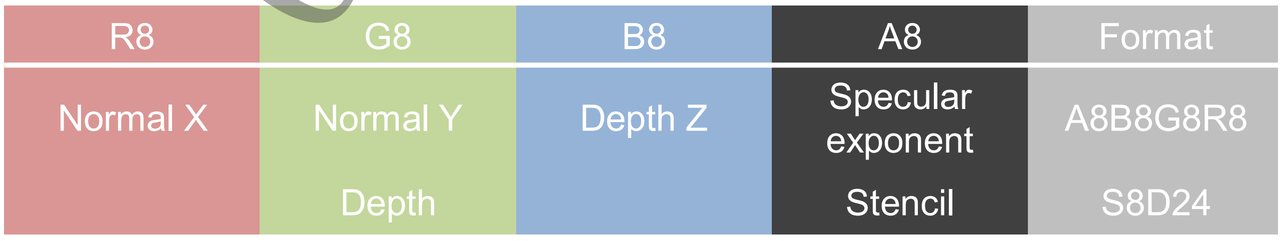
Figure 6　Light Prepass Flow



Building the G-Buffer (Geometry Pass)

The sample program stores the 3xU8 normal and 1xU8 specular exponent value in the G-buffer (equivalent to storing half of the G-buffer from the deferred shading method). This is packed as a 32-bit color surface. During G-buffer sampling, the surface is simply used as a U8U8U8U8-format texture, which can be used as a `half4` type without requiring a instruction for type conversion in the shader.

Figure 7　Light Prepass G-Buffer

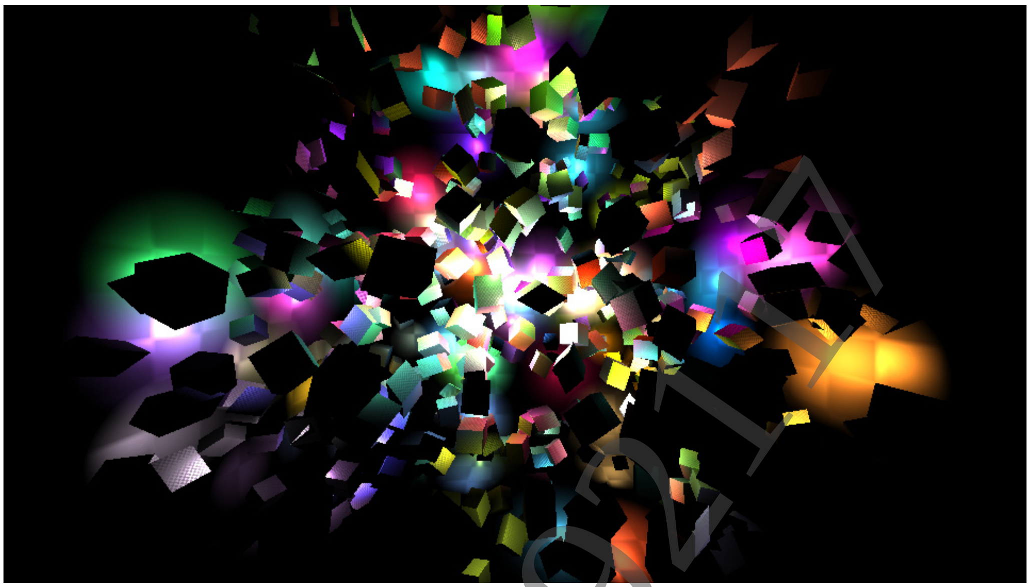


R8	G8	B8	A8	Format
Normal X	Normal Y	Depth Z	Specular exponent	A8B8G8R8
	Depth		Stencil	S8D24

Building the Light Buffer (Lighting Pass)

The sample program uses the F16F16F16F16 format in the lighting buffer. This allows the next pass to maintain high data throughput as it uses the light rendering results compared to the U8U8U8U8 format. The results of rendering the light buffer in the sample program are shown in Figure 8. The contents of the light buffer can also be displayed during execution of the program.

Figure 8 Light Buffer



3 Lighting Techniques

In deferred rendering, rendering efficiency is increased by devising lighting techniques that allow pixels affected by lights to be isolated. In this chapter, two such lighting techniques are described: light volumes and indexed lights. Both deferred shading and light prepass can utilize either of these techniques. During execution of the sample program, it is possible to switch between the techniques from the menu. The number of light sources and their radii can also be adjusted while the program is running.

The sample program calculates the contribution of a light source to a pixel as shown below.

```
N•L * (1 + spec) * lightColor * Attn
```

- N: surface normal
- L: normalized light vector
- spec: specular contribution
- Attn: attenuation

Light Volumes

The source code is in the following file:

- sample_code/graphics/tutorial_deferred_rendering/light_accumulation.cpp

Rendering Light Volumes

In this technique, light sources are rendered with a spherical icosahedron geometry called a light volume. The pixels to be lit can be identified efficiently by rendering only the back faces with the depth-test comparison function set to the opposite of that used in the geometry pass. At that point, the depth buffer must be loaded from memory in order to utilize the depth value results from the geometry pass.

C++ Code to Specify the Depth Test for Efficient Rendering Using Light Volumes

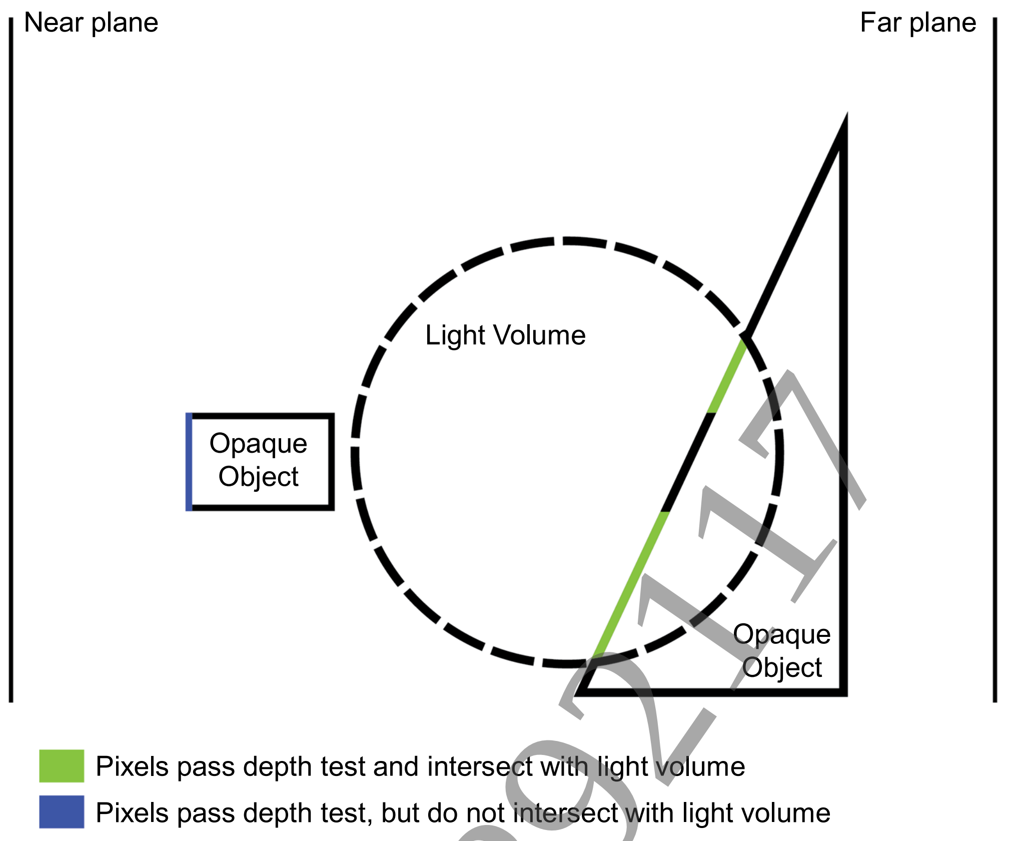
```
sceGxmSetFrontDepthFunc(pContext,SCE_GXM_DEPTH_FUNC_GREATER_EQUAL);
sceGxmSetCullMode(pContext,SCE_GXM_CULL_CW);
```

Figure 9 shows in color the pixels to be lit as a result of the depth test. The blue pixels in the figure pass the depth test even though the entire light volume is located behind them. An optimization technique that addresses this is described in the "Early Out" section of Chapter 4 "Optimization and Performance".

In general, with techniques that use light volumes, the depth test is not performed on the clipped regions when a light volume is clipped by the near or far plane, and the intersection between pixels and lights is not determined correctly. In this case, it is usually necessary to carry out a process to close the sphere. The technique used in the sample program only renders the back faces, so it is not necessary to deal with clipping of light volumes by the near plane, but cases in which light volumes interfere with the far plane must be dealt with. The same is true for the optimization technique which uses the stencil test, described later.

This is an excellent approach for scenes with lots of open spaces, but will never work well for enclosed spaces such as those which are surrounded by walls. The sample program handles only icosahedral light volumes corresponding to point light sources. For actual scenes requiring light sources such as ambient light or spotlights, light volumes with the corresponding shapes would need to be used.

Figure 9 Pixels Lit as a Result of the Depth Test with Respect to a Light Volume


Near plane
Far plane
Light Volume
Opaque Object
Opaque Object
🟩 Pixels pass depth test and intersect with light volume
🟦 Pixels pass depth test, but do not intersect with light volume

Pixel Culling Based on the Stencil Test

The light volume method can be improved by using the stencil test to only render pixels that actually intersect the bounding light volume. The sample program implements this optimization using a depth fail stencil test. For every light source, two draw calls are made. The initial draw serves to stencil mark the lit pixels, and the next draw performs fragment processing on the lighting for only those pixels which were marked by the stencil test. An advantage of this method is that it greatly reduces shader workload, especially if a light source covers only a few pixels. The sample program performs stencil marking in the initial draw using the following steps.

(1) The two-sided stencil mode is used in order to mark the stencils efficiently. Enabling two-sided rendering allows both front and back face operations to be performed at once, so stencils can be marked using a single light volume draw call. Face culling is also disabled so that both sides are fed into the fragment pipeline.

(2) Depth writing is disabled so that the light volume rendering does not affect the results of opaque scene geometry rendering.

(3) When performing stencil marking, the fragment program is disabled so that the fragment shader does not execute. This is significantly faster than using an empty shader.

(4) The comparison function for the depth test is set to `SCE_GXM_DEPTH_FUNC_LESS_EQUAL` for the both faces.

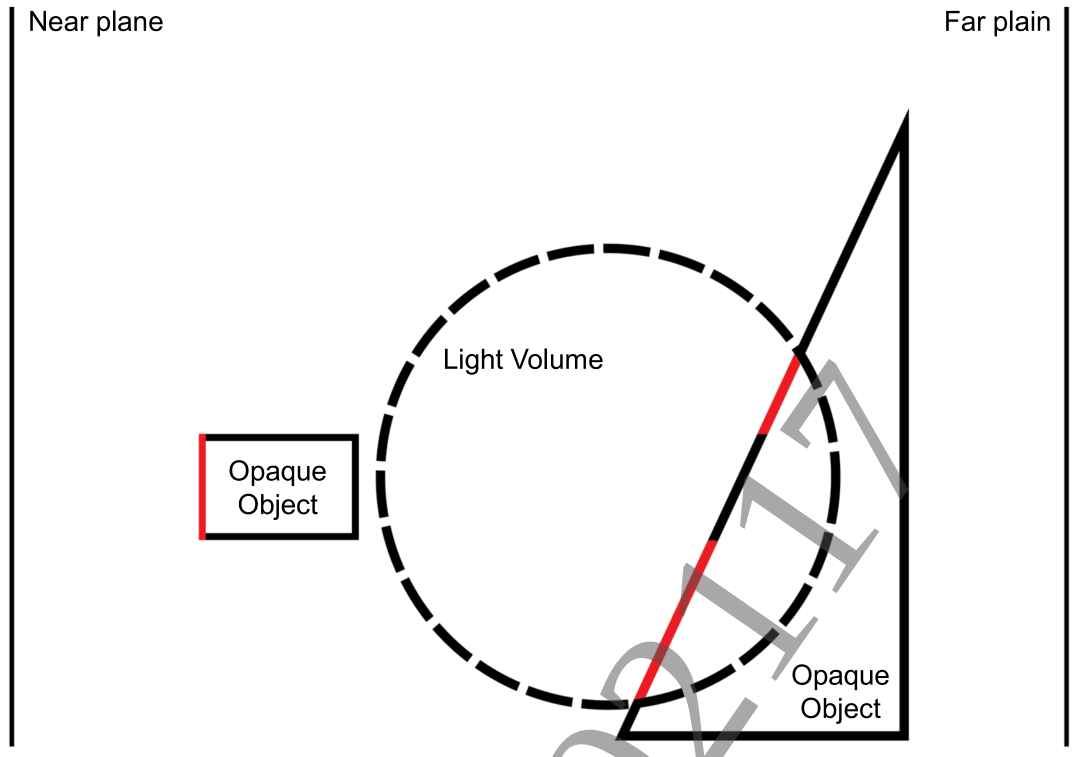
(5) Stencil operation is specified. The stencil test is set to always pass, and for stencil operation when the depth test fails, the back face is set to `SCE_GXM_STENCIL_OP_INCR_WRAP` to increase the stencil value, and the front face is set to `SCE_GXM_STENCIL_OP_DECR_WRAP` to decrease the stencil value. Note that stencil operation is also specified such that the stencil value is set to 0 if the depth test succeeds for a back face, and is left unchanged if the depth test succeeds for a front face.

After these states have been specified, the light volumes are rendered. Figure 10 and Figure 11 show the resulting stencil values based on the depth tests on the back and front faces respectively. If a pixel intersects a bounding light volume, the front face passes the depth test and the back face fails the depth test, so based on this stencil operation, the pixel to be lit is marked with a stencil value of 1 (the red pixels in Figure 11). If the light volume is located in front of the pixel, both the front face and the back face pass the depth test, so the stencil value of 0 is left unchanged. If instead the light volume is located behind the pixel, neither face passes the depth test, so the stencil value of the back face is incremented, and the stencil value of the front face is decremented and ultimately reverts to 0 (the blue pixels in Figure 11).

C++ Code to Set the Stencil Culling State (Step 1)

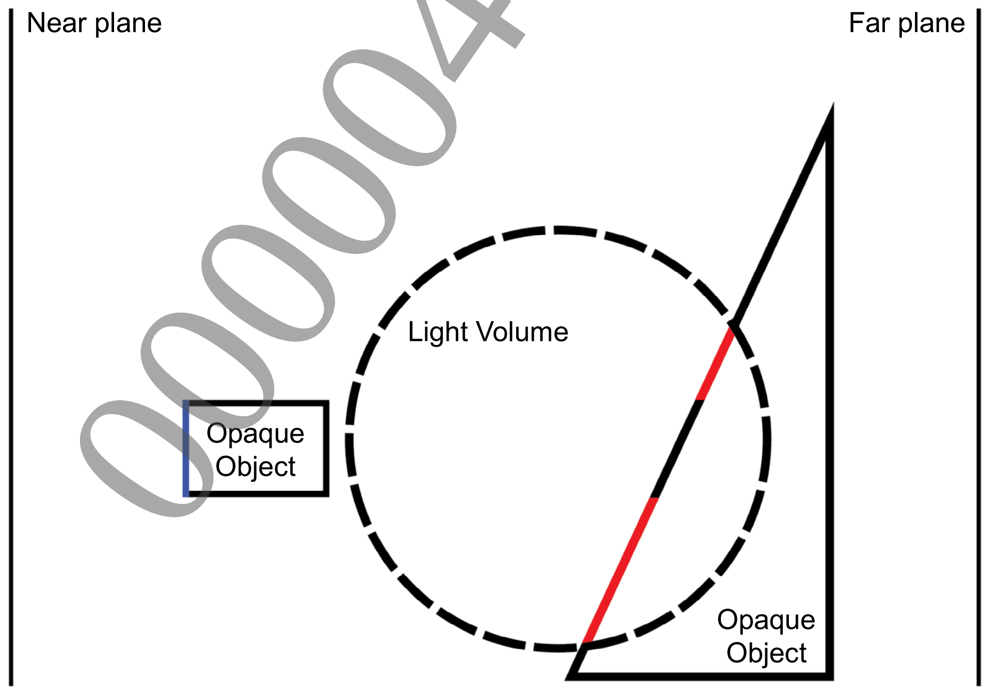
```
// Enable two-sided rendering
sceGxmSetTwoSidedEnable(pContext, SCE_GXM_TWO_SIDED_ENABLED);
// Disable face culling since both sides are needed to perform the stencil
operation
sceGxmSetCullMode(pContext, SCE_GXM_CULL_NONE);

// Disable depth value update
sceGxmSetFrontDepthWriteEnable(pContext, SCE_GXM_DEPTH_WRITE_DISABLED);
sceGxmSetBackDepthWriteEnable(pContext, SCE_GXM_DEPTH_WRITE_DISABLED);

// Disable fragment programs during the stencil update stage
sceGxmSetFrontFragmentProgramEnable(
    pContext, SCE_GXM_FRAGMENT_PROGRAM_DISABLED);
sceGxmSetBackFragmentProgramEnable(
    pContext, SCE_GXM_FRAGMENT_PROGRAM_DISABLED);

// Set the depth test comparison function for both faces to
SCE_GXM_DEPTH_FUNC_LESS_EQUAL
sceGxmSetFrontDepthFunc(pContext, SCE_GXM_DEPTH_FUNC_LESS_EQUAL);
sceGxmSetBackDepthFunc(pContext, SCE_GXM_DEPTH_FUNC_LESS_EQUAL);

// Increase the count for pixels located in front of the back faces
sceGxmSetBackStencilFunc(
    pContext, SCE_GXM_STENCIL_FUNC_ALWAYS, SCE_GXM_STENCIL_OP_ZERO,
    SCE_GXM_STENCIL_OP_INCR_WRAP, SCE_GXM_STENCIL_OP_ZERO, 0xff, 0xff);

// Decrease the count for pixels located in front of the front faces
// The result will be that the count for all pixels located within a light volume
will be 1
sceGxmSetFrontStencilFunc(
    pContext, SCE_GXM_STENCIL_FUNC_ALWAYS, SCE_GXM_STENCIL_OP_KEEP,
    SCE_GXM_STENCIL_OP_DECR_WRAP, SCE_GXM_STENCIL_OP_KEEP, 0xff, 0xff);

// Render the light volumes
sceGxmDraw(
    pContext, SCE_GXM_PRIMITIVE_TRIANGLES,SCE_GXM_INDEX_FORMAT_U16,
    passData->indexBuffer, 60);
```

Figure 10 Stencil Values Resulting from the Depth Test with Respect to the Light Volume Back Face



🟥 Pixels do not pass depth test and stencil incremented by 1 (stencil = 1)

Figure 11 Stencil Values Resulting from the Depth Test with Respect to the Light Volume Front Face



🟥 Pixels pass depth test and stencil is unchanged (stencil = 1)
🟦 Pixels do not pass depth test and stencil decreased by 1 (stencil = 0)

Actual lighting is performed by the second light volume draw call. In this draw call, a stencil test is performed which checks whether or not the stencil value is 1. Based on this, the lighting shader is executed on only the pixels which were stencil marked in the previous call. The steps carried out by the second draw are as follows.

(1) Set the stencil reference value of the front and back faces to 1.

(2) Set the stencil comparison functions of the front and back faces to `SCE_GXM_STENCIL_FUNC_EQUAL` and set the operation of all stencils to `SCE_GXM_STENCIL_OP_ZERO`. Set the depth test for both faces to always pass so that the stencil values will always be reverted to 0.

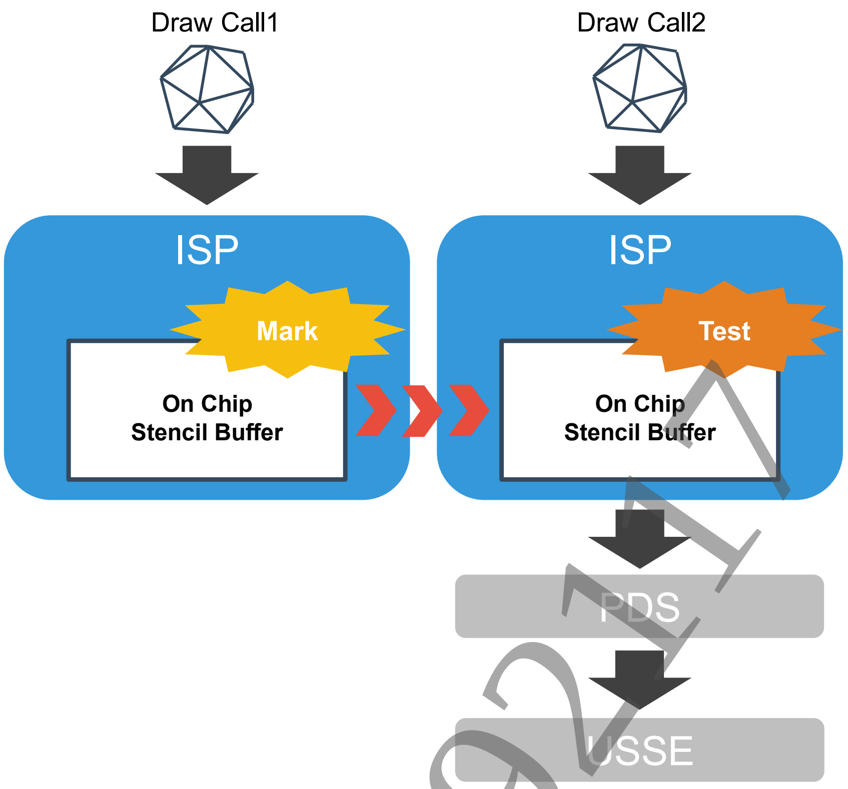
(3) Enable the fragment programs, which were disabled for the initial draw call.

As a result of these state settings, the stencil values are marked to 1, and shading is executed only once for pixels that are exposed to light. In the GPU of PlayStation®Vita, the stencil test is performed on the buffer in the ISP unit, which precedes the PDS unit and USSE unit, so there are no interactions with external memory and the test executes quickly (Figure 12). In addition, since the stencil test is executed prior to fragment processing, it is possible to efficiently send only the pixels which are to be lit to the shader unit.

C++ Code to Set the Stencil Culling State (Step 2)

```
sceGxmSetFrontStencilRef(pContext, 0x1);
sceGxmSetBackStencilRef(pContext, 0x1);

sceGxmSetBackStencilFunc(pContext,SCE_GXM_STENCIL_FUNC_EQUAL,
    SCE_GXM_STENCIL_OP_ZERO,SCE_GXM_STENCIL_OP_ZERO,
    SCE_GXM_STENCIL_OP_ZERO,0xff,0xff);

sceGxmSetFrontStencilFunc(pContext,SCE_GXM_STENCIL_FUNC_EQUAL,
    SCE_GXM_STENCIL_OP_ZERO,SCE_GXM_STENCIL_OP_ZERO,
    SCE_GXM_STENCIL_OP_ZERO,0xff,0xff);

// Disable depth test
sceGxmSetBackDepthFunc(pContext,SCE_GXM_DEPTH_FUNC_ALWAYS);
sceGxmSetFrontDepthFunc(pContext,SCE_GXM_DEPTH_FUNC_ALWAYS);

// Enable fragment programs and set the lighting shader
sceGxmSetBackFragmentProgramEnable(pContext,
    SCE_GXM_FRAGMENT_PROGRAM_ENABLED);
sceGxmSetFrontFragmentProgramEnable(pContext,
    SCE_GXM_FRAGMENT_PROGRAM_ENABLED);

// Render the light volumes
sceGxmDraw(pContext,SCE_GXM_PRIMITIVE_TRIANGLES,
    SCE_GXM_INDEX_FORMAT_U16,passData->indexBuffer,60);
```

Figure 12 Flow of Culling using Stencils



Indexed Lights

The indexed light technique is a technique in which a loop in the fragment shader is repeated once for each light. It is also sometimes used in forward rendering as well. In this technique, light rendering for all light sources can be processed in a single shader call.

The source code is in the following file:

- sample_code/graphics/tutorial_deferred_rendering/single_pass_lighting.cpp

Tile-Based Light Culling

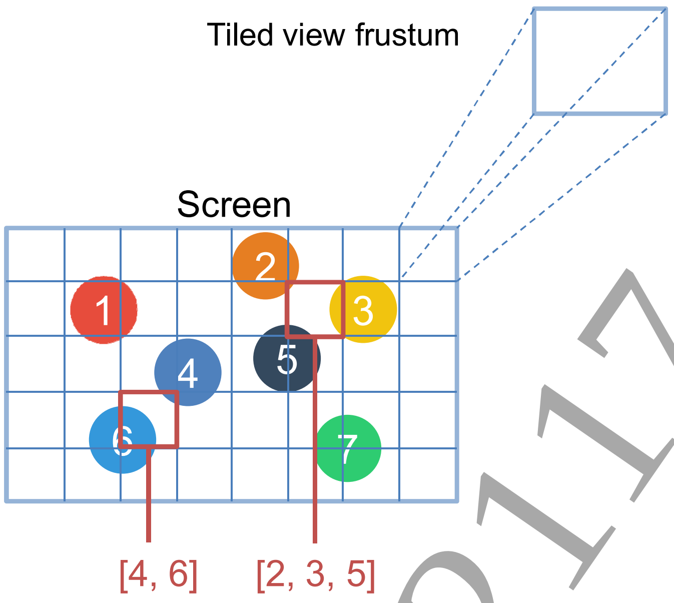
In the indexed light technique, the scene is partitioned into multiple tiles as shown in Figure 13, and for each tile, all light sources that can affect the tile are determined beforehand by the CPU, thus reducing lighting calculation time. In the lighting pass, lighting is calculated by referencing a list of only the contributing light sources. In this tutorial, this is called "combined lighting." The size of the tiles on which culling is performed is the same as the GPU of PlayStation®Vita hardware tile size: 32 x 32. Culling is performed at the granularity of tiles, so a given pixel within a tile may not intersect with a light in the list which is created. In such cases, performance can be improved by applying the optimization described in the "Early Out" section of Chapter 4 "Optimization and Performance".

The drawbacks of this method include the high CPU cost for determining whether or not light sources affect multiple tiles, the dynamic branching that occurs in the shader, and the fact that, because the stencil buffer cannot store volume information from more than one light source, the GPU’s depth test cannot be used to determine whether or not light sources can be excluded from the computation.

An advantage is that the G-buffer need only be referenced once per pixel, regardless of the number of light sources that contribute to a pixel.

The same is true for any lighting computations that are independent of the light position, such as view vector normalization, or multiplication by albedo factor. There is also no limit to how individual contributing light sources can be combined. For example, one can do gamma correction at the end of the shader, or pull out common factors.

Figure 13 Screen-Space Tile Projected View Volume



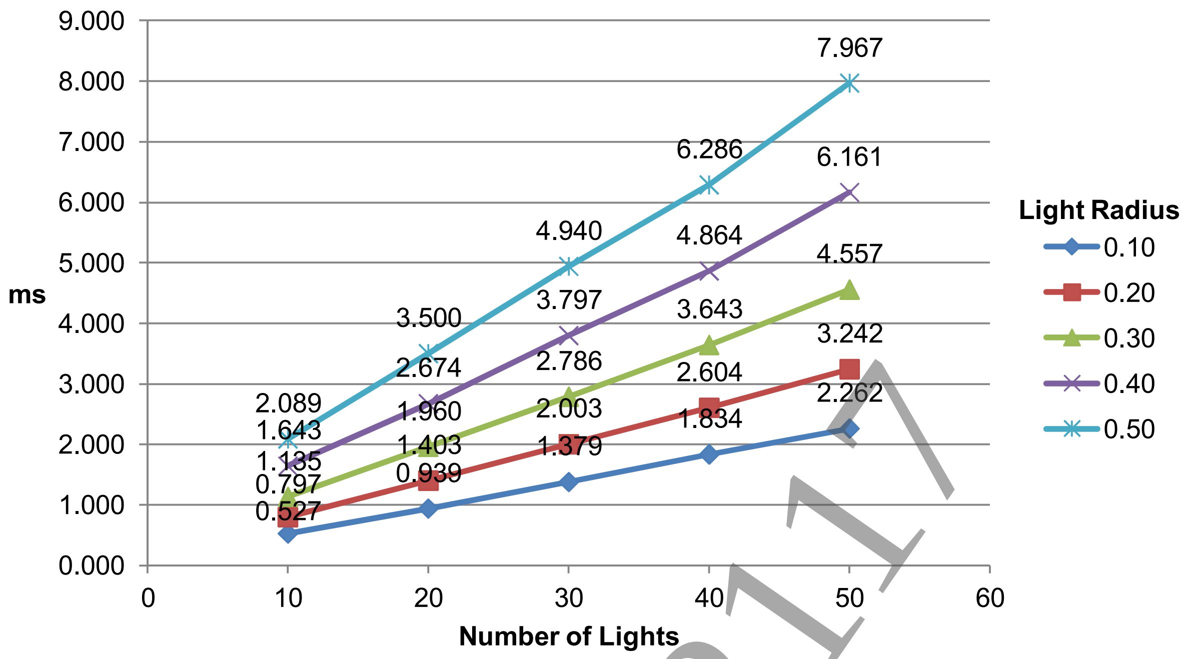
Interference between a light and a tile frustum is determined in the same manner as when a view frustum is used to frustum culling. The equation for each face of a frustum defined by a tile is solved by calculating the surface normal as the cross-product of the vectors from the origin to the corners of the tile. By comparing the distances from the center of a light to each face, one can determine whether there is interference between the light and the face.

C++ Code to Calculate the Equations for the Surfaces of the Frustum Created by Each Tile

```
const Aos::Vector3 yStep = *viewSpaceDeltaY / (horizontalPlanes - 1);
Aos::Vector3 left = *viewSpaceLowerLeft + *viewSpaceDeltaY;
for(uint32_t y = 0; y < horizontalPlanes; ++y)
{
    const Aos::Vector3 right = left + *viewSpaceDeltaX;
    const Aos::Vector3 n = Aos::normalize(Aos::cross(right,left));
    Aos::Vector4 plane = Aos::Vector4(n);
    plane.setW(0);
    planes[y] = plane;
    left -= yStep;
}

const Aos::Vector3 xStep = *viewSpaceDeltaX / (verticalPlanes - 1);
Aos::Vector3 lower = *viewSpaceLowerLeft;
for(uint32_t x = 0; x < verticalPlanes; ++x)
{
    const Aos::Vector3 upper = lower+ *viewSpaceDeltaY;
    const Aos::Vector3 n = Aos::normalize(Aos::cross(lower,upper));
    Aos::Vector4 plane = Aos::Vector4(n);
    plane.setW(0); planes[x + horizontalPlanes] = plane;
    lower += xStep;
}
```

Figure 14 graphs the time required to perform culling as a function of the number of lights and the radius of the lights. Processing time increases linearly as the number of lights and the light radius increases.

Figure 14 CPU Time Consumed by Tile-Based Light Culling



Per-Tile Rendering with Region Clip

Per-tile rendering is performed based on the list created by light culling. This is implemented in the sample program using the region clip feature. The region of each tile is set by the coordinates of the rectangular region specified by the region clip, the clip mode is set so that all tiles outside that region are clipped, and then the full-screen rectangle is rendered. This is repeated for all tiles in the screen space. The position and color information for the lights in the list are assigned as an array in a user-defined uniform buffer, and the size of this array (i.e., the number of lights for which there is interference) is also passed via the uniform buffer such that it can be referenced from the lighting shader.

C++ Code to Render Each Tile Using Region Clip

```
for(uint32_t tile = 0; tile < passData->tileCount; ++tile)
{
    ...
    // Set the clip
    sceGxmSetRegionClip(pContext, SCE_GXM_REGION_CLIP_OUTSIDE,
        left, top, right, bottom);
    // Render full-screen quad
    sceGxmDraw(data->context, SCE_GXM_PRIMITIVE_TRIANGLES,
        SCE_GXM_INDEX_FORMAT_U16, indexBuffer, 3);
}
```

4 Optimization and Performance

In this chapter, optimization techniques are explained, and the change in performance when these optimizations are applied with the lighting techniques described in Chapter 3 "Lighting Techniques", are shown in graphical form.

Depth Resolve

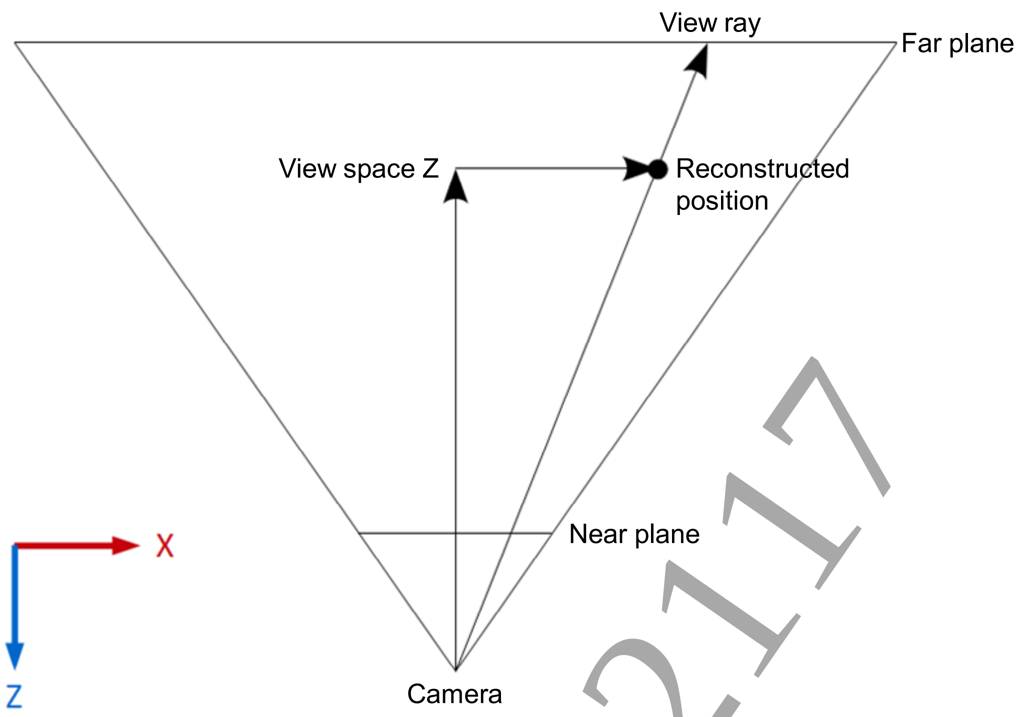
In order to perform lighting calculations, the position of each pixel in the space in which lighting is performed, as well as in the world space or view space, is required. In the GPU of PlayStation®Vita, G-buffers of more than 64 bits cannot be created due to limitations in the color surface and the output register, so pixel positions cannot normally be included in a G-buffer. For this reason, pixel positions must be reconstructed in passes other than the geometry pass.

In order to reconstruct pixel positions, the Z-buffer is generally used. The view space depth value is reconstructed from the screen-space depth value by referencing the Z-buffer output during the geometry pass as a texture and using the view port transform and projection transform coefficients from the geometry pass. By using the ratio between this reconstructed depth value and the near plane depth value, and scaling the view ray to the near plane, the XY-coordinates of the pixel in the view space can be reconstructed (Figure 15). In this tutorial, reconstructing the pixel position in this way is called "Depth Resolve". Since the sample program performs lighting in the view space, it reconstructs pixel positions in the view space. The reconstructed pixel coordinates are stored in an F16F16F16F16 format color surface, and are referenced as a texture in the lighting pass.

Cg Code to Perform Depth Resolve

```
#define NEAR_PLANE 0.01
#define FAR_PLANE 10.0

#define DEPTH_UNPACK_A ((-2 * FAR_PLANE * NEAR_PLANE) / (NEAR_PLANE - FAR_PLANE))
#define DEPTH_UNPACK_B ((FAR_PLANE + NEAR_PLANE) / (NEAR_PLANE - FAR_PLANE))

real4 getPositionAndInvDistance(float4 texcoord, float3 nearPlaneRay)
{
    const float depth =
        (DEPTH_UNPACK_A/NEAR_PLANE)/
        (1 - DEPTH_UNPACK_B - 2*tex2Dproj<float>(depthTexture,texcoord.xyw));
    float3 pos = nearPlaneRay*depth;
    float4 result = float4(pos,1/sqrt(dot(pos,pos)));
    return result;
}
```

Figure 15 Depth Resolve



Depth Resolve can refer to a method in which an entire screen is processed in a single independent pass, or to a method in which each lit pixel is processed in the lighting pass. It is possible to switch between these from the menu in the sample program. When Depth Resolve is set to ON, processing is performed in a single independent pass, and when it is set to OFF, processing is performed in the lighting pass.

When selecting whether to perform Depth Resolve in a separate pass or in the lighting pass, there is a trade-off between processing time and memory usage. A benefit of performing Depth Resolve in the lighting pass is that it reduces memory usage, since position data does not need to be retained as a texture after Depth Resolve. The graphs in Figure 16 and Figure 17 show increases in fragment shader processing time for various lighting techniques when Depth Resolve is performed in the lighting pass of the deferred shading method. Figure 18 and Figure 19 show corresponding graphs for light prepass.

In the graphs, the lighting techniques shown are as follows:

- LVS: Light volume technique with the stencil test
- LV: Light volume technique without the stencil test
- LI: Indexed light technique with tile-based light culling

In the graphs, "Single Pass" shows processing time when Depth Resolve is performed in a single pass. The time required to reconstruct pixel positions for the entire screen in a single independent pass is a constant value regardless of the number and size of the lights. In contrast, when Depth Resolve is performed in the lighting pass, Depth Resolve is performed on each lit pixel, so the time consumed by Depth Resolve increases as the number and radius of the lights increases. For scenes in which pixels are lit by multiple lights and light volume techniques are used, Depth Resolve is performed on each light and is thus extremely inefficient.

Since the sample program does not take ambient lighting into consideration, only pixels lit by point light sources are considered. As a result, for light volume techniques, the times required to perform Depth Resolve in the lighting pass are lower than when performing Depth Resolve in a single pass when the number and radius of the lights is small. However, in real scenes where ambient lighting is taken into consideration, lighting which uses full-screen quads is usually required, so performing Depth Resolve in a single independent pass is recommended if memory allows. For the indexed lighting technique, since Depth Resolve is performed once per pixel, there is no performance degradation even if the number and radius of the lights are increased.

Figure 16 Deferred Shading -
Fragment Processing Time Increases when Depth Resolve is Performed in the Lighting Pass

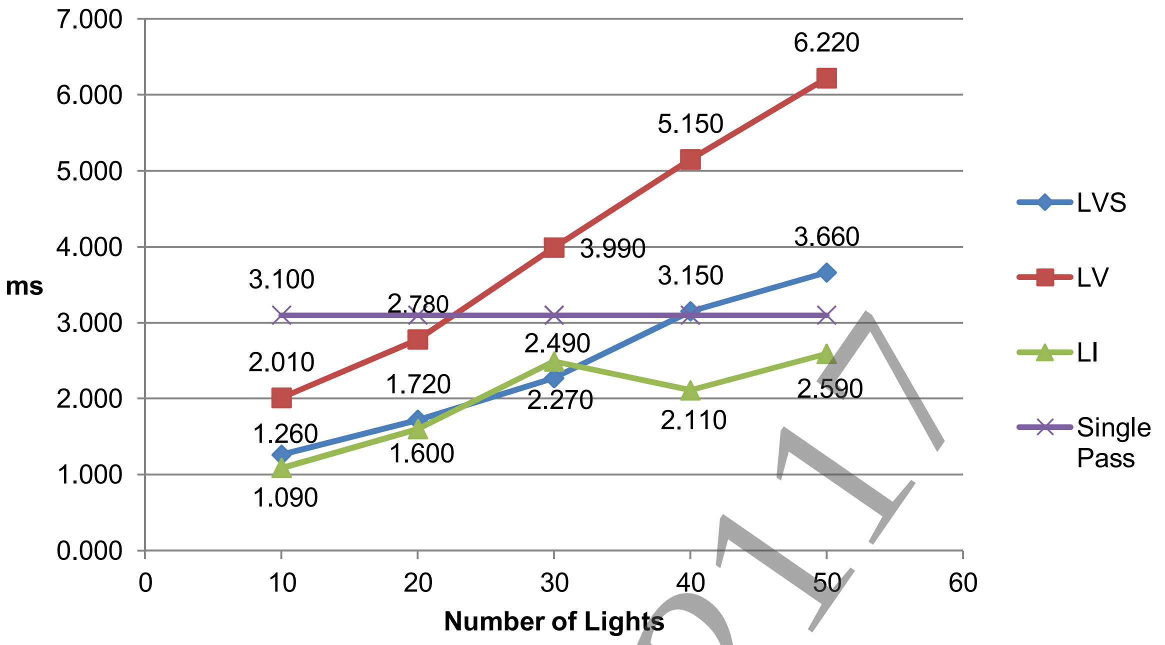


Figure 17 Deferred Shading -
Fragment Processing Time Increases when Depth Resolve is Performed in the Lighting Pass

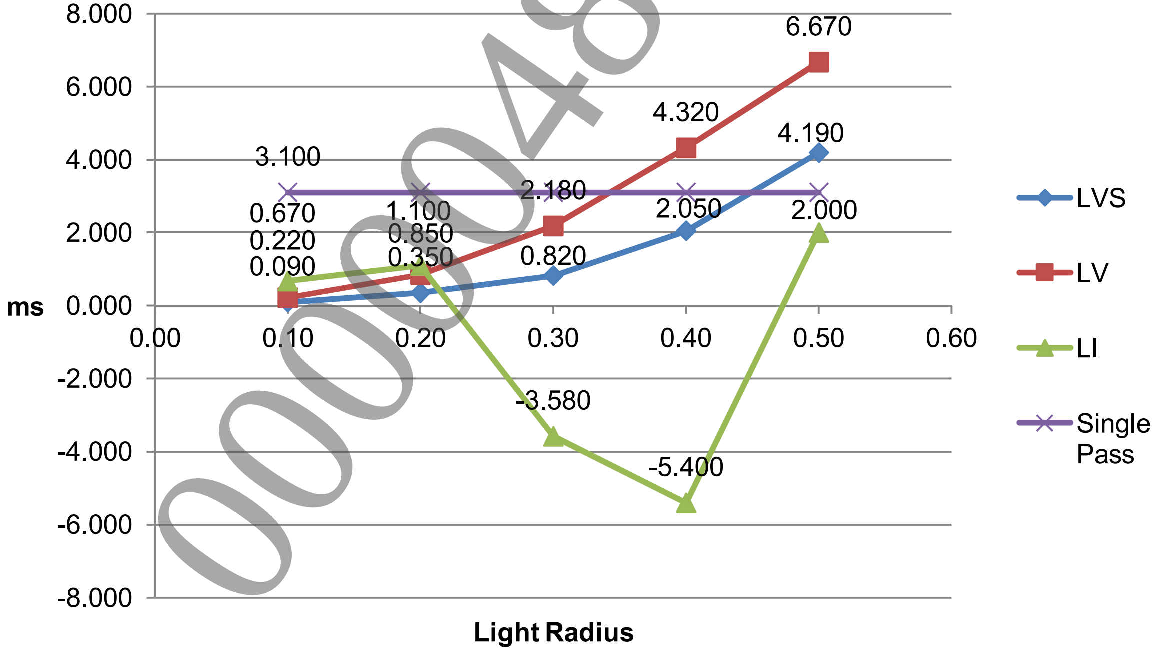


Figure 18　Light Prepass -
Fragment Processing Time Increases when Depth Resolve is Performed in the Lighting Pass

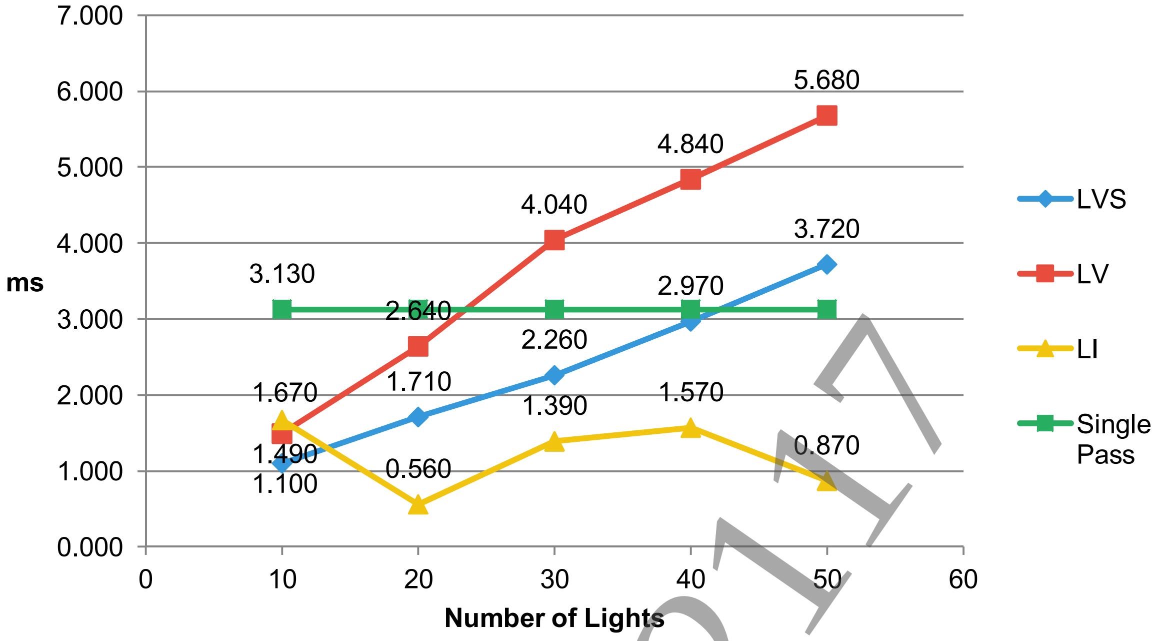
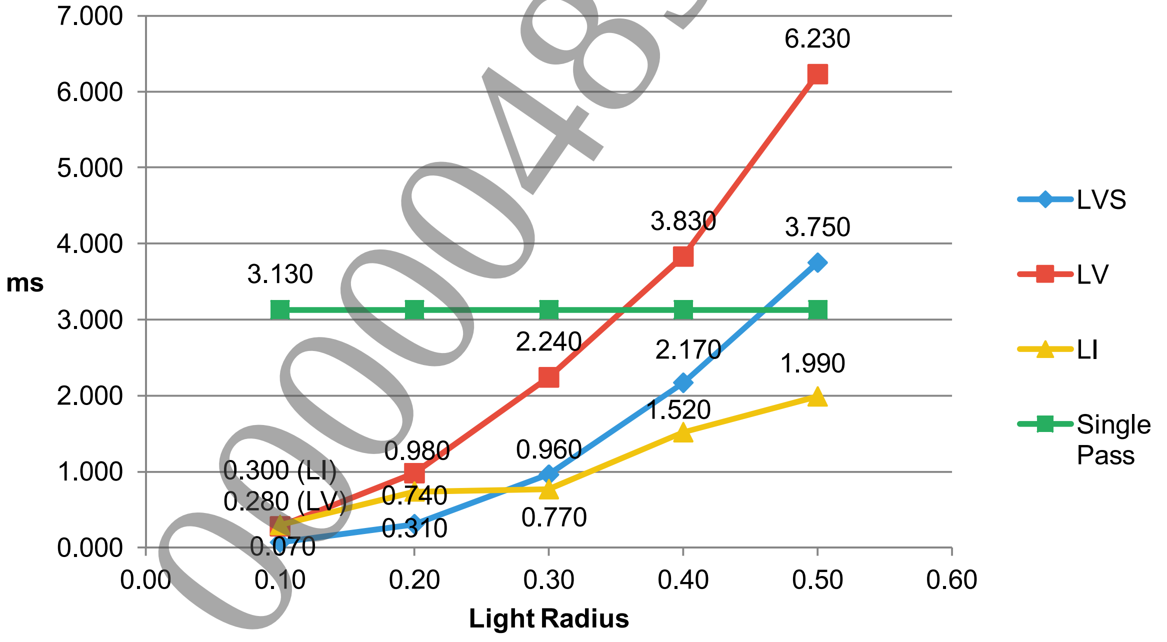


Figure 19　Light Prepass -
Fragment Processing Time Increases when Depth Resolve is Performed in the Lighting Pass



For the graphs with "Number of Lights" on the horizontal axis, the light radius is 0.45. For the graphs with "Light Radius" on the horizontal axis, the calculations are performed with 40 lights.

Aside from performing Depth Resolve, method which outputs pixel position to the writable user-defined uniform buffer in the geometry pass is also possible. The graphs in Figure 20 and Figure 21 compare the performance of such a method with performing Depth Resolve in a single independent pass. In the graphs, "Memory Export" represents the method which outputs to a uniform buffer during the geometry pass. The type of the uniform buffer is set to `half4` to match that of the color buffer for Depth Resolve.

The time required for the geometry pass when outputting to the uniform buffer is approximately twice the total processing time for the geometry pass and the pass to perform reconstruction in Depth Resolve. For this reason, using Depth Resolve to reconstruct pixel positions in a separate pass from the geometry pass is recommended over using a uniform buffer to store pixel positions in a G-buffer in the geometry pass. Increases in performance cost should always be considered carefully when expanding the G-buffer using a writable uniform buffer, for data other than pixel position as well. Since the size of the G-buffer with light prepass is half of that with deferred rendering, the time required for the geometry pass is slightly lower for light prepass than for deferred shading.

Figure 20 Deferred Shading -
Fragment Processing Time for Single-Pass Depth Resolve and Uniform Buffer Output

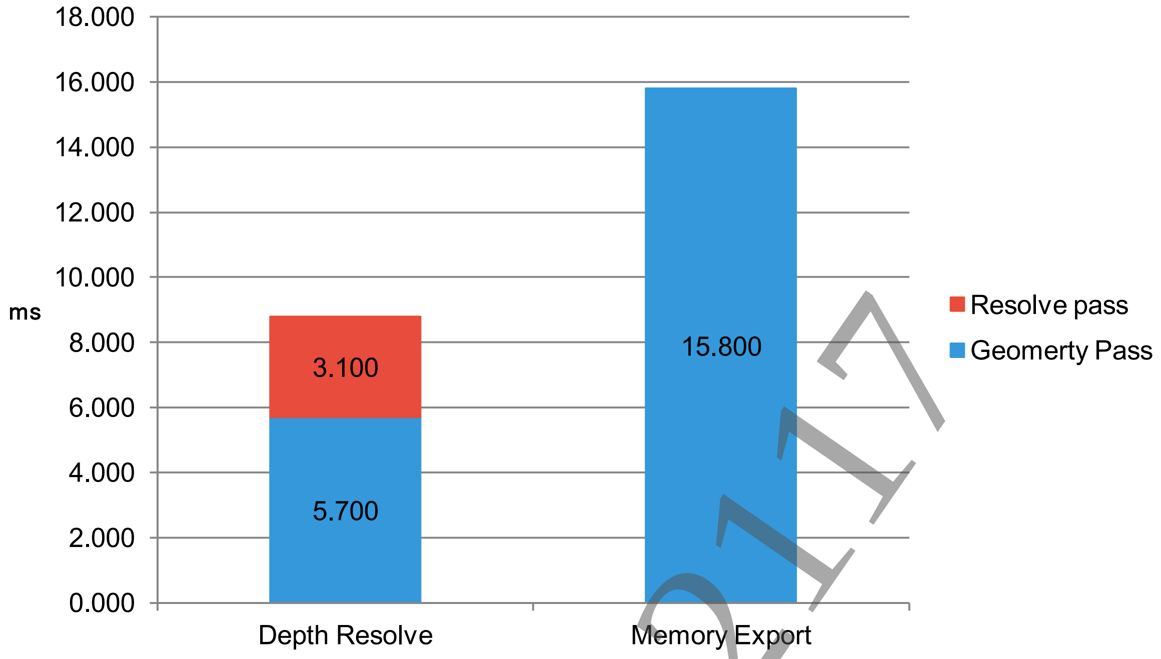
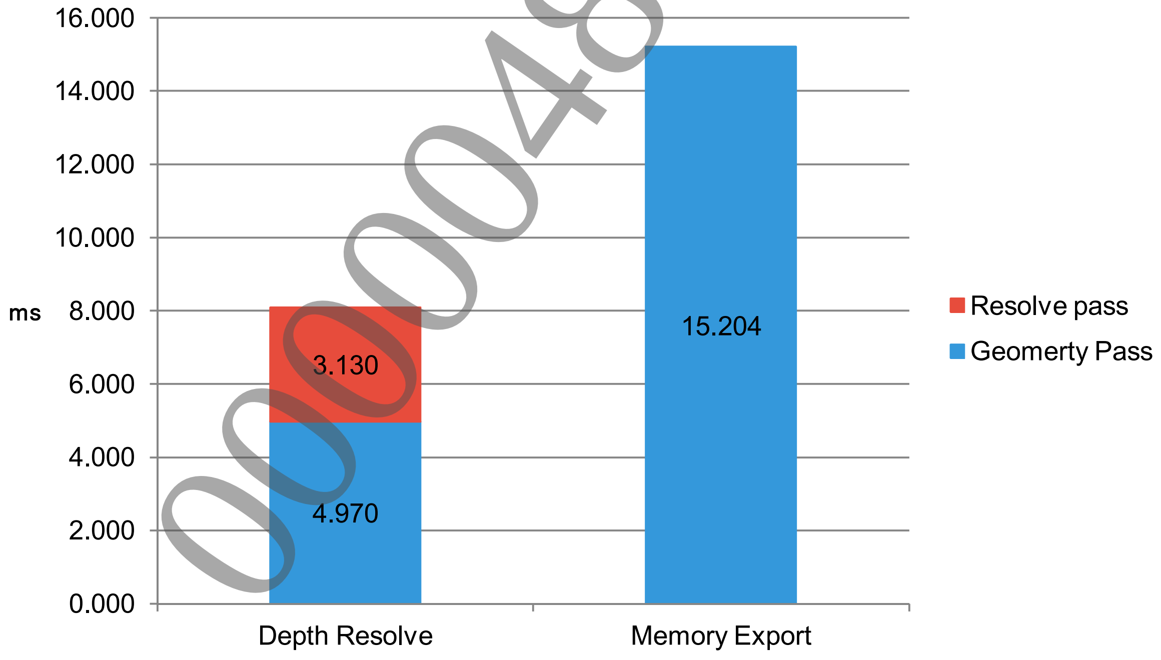


Figure 21 Light Prepass -
Fragment Processing Time for Single-Pass Depth Resolve and Uniform Buffer Output



Split G-Buffer

With deferred shading, instead of referencing the G-buffer as a 64-bit U32U32-format texture and unpacking it in the fragment shader, it can be referenced as two 32-bit U8U8U8U8-format textures, each being either a diffuse color texture or a normal texture, reducing the processing cost of unpacking, and furthermore normalization can be delegated to the texture unit. In this tutorial, this is called "Split G-Buffer". Enabling Split G-Buffer, not only allows diffuse colors and normals to be used as-is, but gamma correction can also be performed. It is possible to enable and disable Split G-Buffer from the menu of the sample program while it is running.

With Split G-Buffer, texture words are initialized by specifying the format to be `SCE_GXM_TEXTURE_FORMAT_U8U8U8U8_ABGR` and the width to be twice that of the G-buffer. Here, the base address of the G-buffer is specified for the first texture, but the address which is 4 bytes away from the starting address of the G-buffer is specified for the second texture so that the high-order 32 bits of the normal stored is at the starting address. As a result, the diffuse color and normal can each be referenced at the same texture coordinate.

C++ Code to Initialize Two Textures for Split G-Buffer

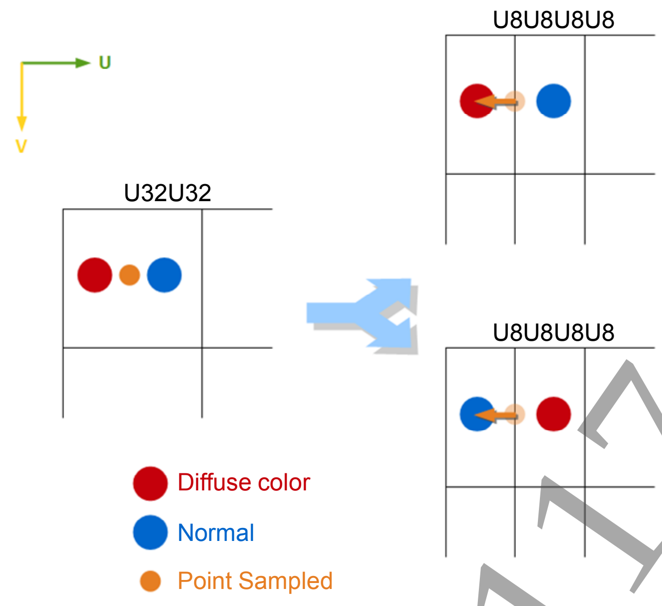
```
CreateTextureFromAddress(
    &data->splitGbuffer0,
    data->gbuffer,
    SCE_GXM_TEXTURE_FORMAT_U8U8U8U8_ABGR,
    data->gBufferWidth*2,
    data->gBufferHeight,
    SCE_GXM_TEXTURE_LINEAR);

CreateTextureFromAddress(
    &data->splitGbuffer1,
    ((char*)data->gbuffer) + 4,
    SCE_GXM_TEXTURE_FORMAT_U8U8U8U8_ABGR,
    data->gBufferWidth*2,
    data->gBufferHeight,
    SCE_GXM_TEXTURE_LINEAR);
```

If this texture, which is twice as wide as the render target, is referenced as is, interpolated texture coordinates will fall between texels. Thus, for the fragment shader to sample diffuse color or normal texels from the respective textures, the vertex shader shifts the UV coordinates to the left by a half-texel (Figure 22).

Cg Code to Shift the UV Coordinates for Split G-Buffer

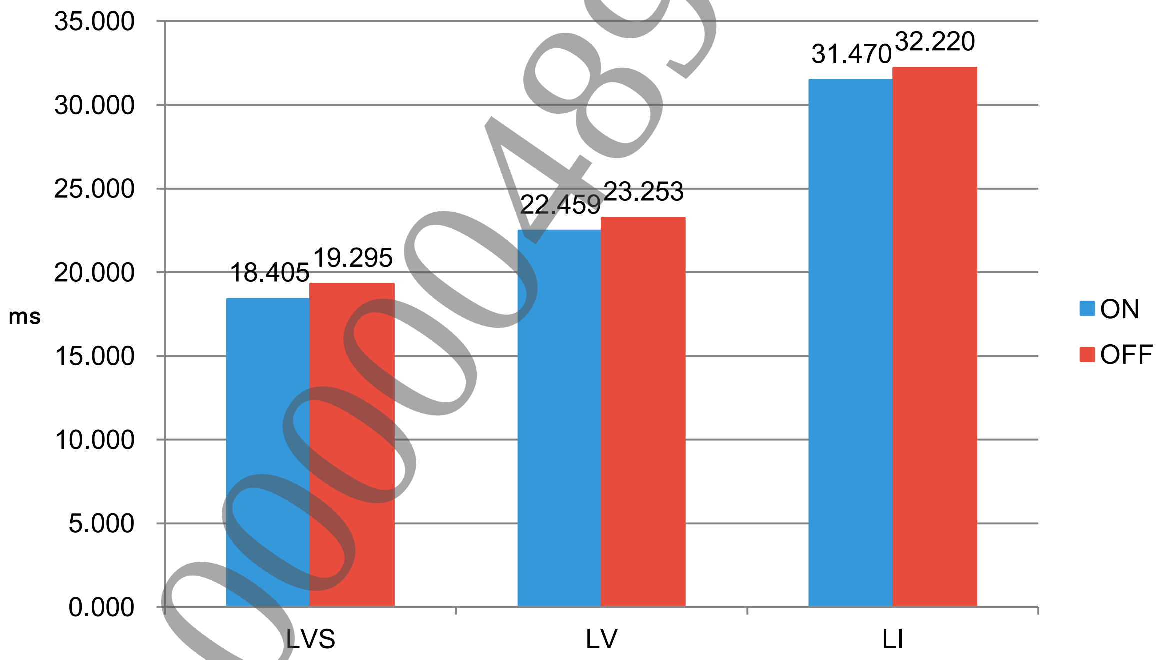
```
vTexCoord.x  = clipSpacePosition.x * 0.5f + (0.5f - 0.25f /
          GRAPHICS_UTIL_DEFAULT_DISPLAY_WIDTH) * clipSpacePosition.w;
vTexCoord.y  = (clipSpacePosition.w - clipSpacePosition.y) * 0.5f;
vTexCoord.zw = clipSpacePosition.w;
```

Figure 22 Split G-Buffer



The graph in Figure 23 compares the lighting pass processing time when Split G-Buffer is enabled and disabled. By enabling Split G-Buffer, processing time can be reduced with any lighting technique.

Figure 23 Deferred Shading - Lighting Pass Processing Time when Split G-Buffer is Enabled or Disabled



In the graph, the lighting techniques shown are as follows:

- LVS: Light volume technique with the stencil test
- LV: Light volume technique without the stencil test
- LI: Indexed light technique with tile-based light culling

Early Out

In the light volume technique without the stencil test, if an entire light volume is located in front of a pixel then the depth test will not pass and the pixel will not be lit, but if the entire light volume is located behind the pixel then the depth test will pass and fragment processing will proceed, despite the fact that the pixel is not located within the light volume. Furthermore, with the indexed light technique, tile-based light culling can only determine intersections at the granularity of tiles, so lighting calculations are applied to some pixels which do not actually fall within the light. (In such cases, shading results can ultimately be corrected by taking light attenuation into account.) To perform light processing efficiently, an optimization is performed which compares the distance from a pixel to the center of a light with the radius of the light in order to determine whether the light contributes to the pixel before doing lighting calculations, and if the light does not contribute to the pixel, the light processing is discontinued. In this tutorial, this is called "Early Out." It is possible to switch between enabling and disabling Early Out from the menu in the sample program while it is running.

In determining the intersection between a pixel and a light, the amount of computation can be reduced by comparing the distance between the pixel and the center of the light with the radius of the light using the squares of each.

Cg Code to Perform Early Out

```
const real3 distance_vector = PosAndRadius.xyz-pixel_position.xyz;
const real square_distance = dot(distance_vector,distance_vector);
const real square_radius = PosAndRadius.w;
real4 result = 0;

#ifndef NO_EARLY_OUT_LIGHTS
// we use a condition to check distance even for bounding volume
// for stencil as we render a icosahedron which encomposses the light
if(square_distance <= square_radius)
#endif
{
    // Perform lighting calculation here
}
```

When Early Out is enabled, the shader will operate in "Per-instance" mode because of branch creation, and can reduce the light processing time in many cases. The graphs in Figure 24 and Figure 25 illustrate the differences in fragment processing time in the lighting pass for each lighting technique when Early Out is enabled and disabled (subtracting the processing time when enabled from the processing time when disabled). Meanwhile, Figure 26 and Figure 27 are the analogous graphs for light prepass. In the light volume technique using the stencil test, pixels to be lit are determined based on the stencil test, so Early Out is not effective. In contrast to this, for the light volume technique without the stencil test and the indexed light technique, the effect of enabling Early Out is readily apparent.

Figure 24 Deferred Shading -
Difference in Lighting Pass Fragment Processing Time when Early Out is Enabled or Disabled

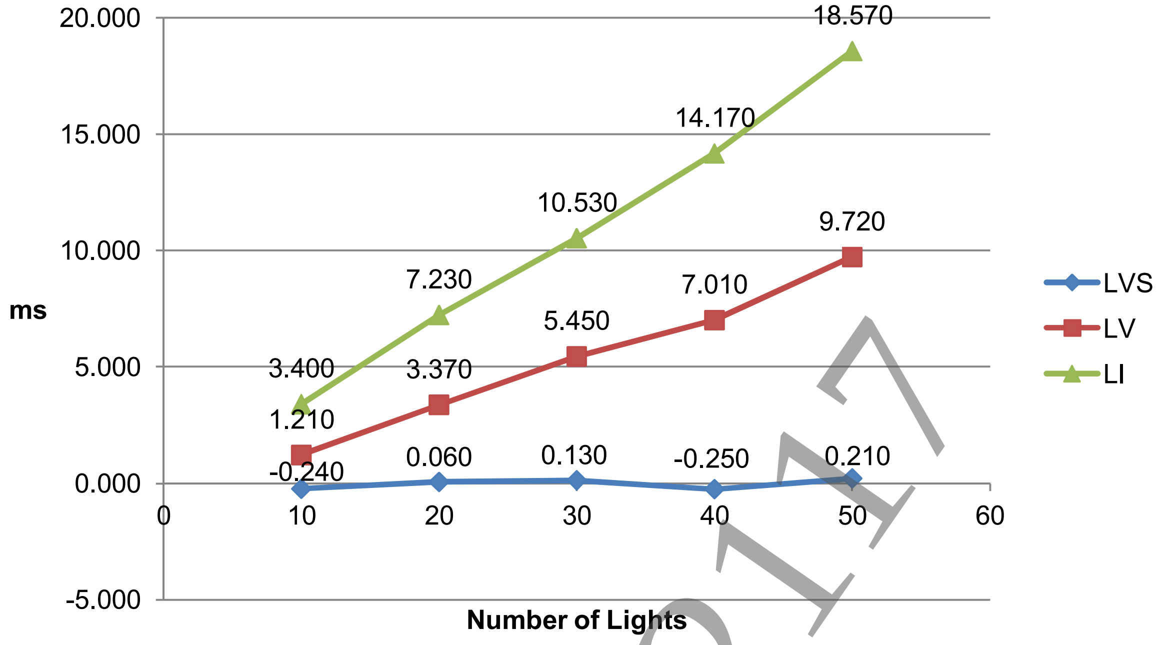


Figure 25 Deferred Shading -
Difference in Lighting Pass Fragment Processing Time when Early Out is Enabled or Disabled

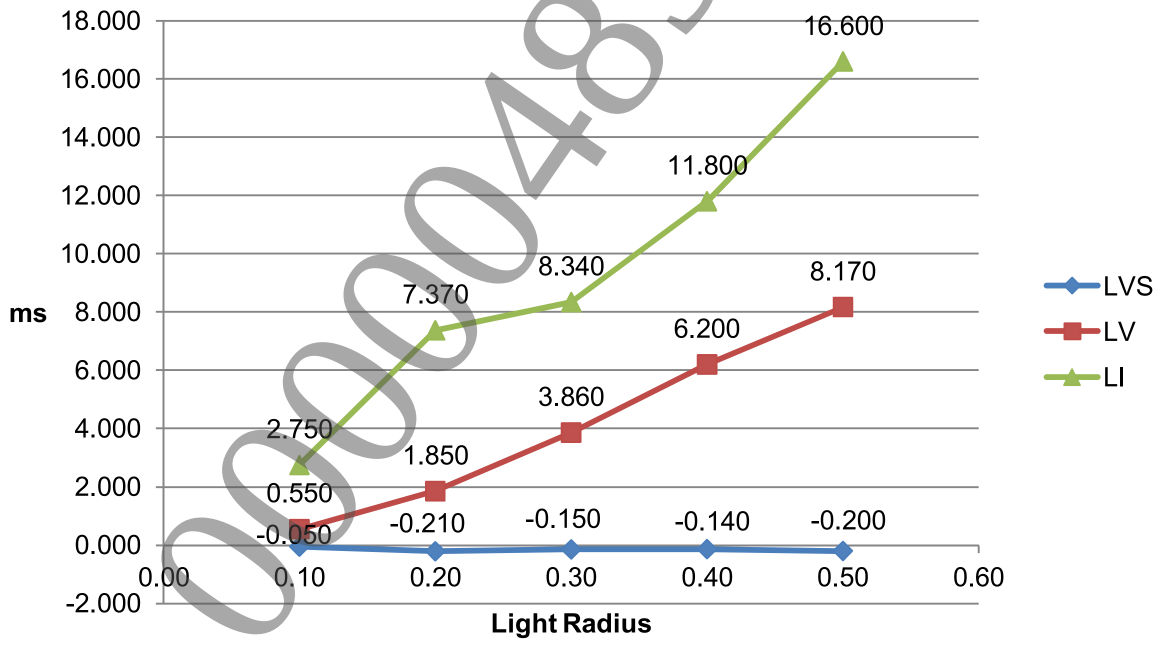


Figure 26 Light Prepass -
Difference in Lighting Pass Fragment Processing Time when Early Out is Enabled or Disabled

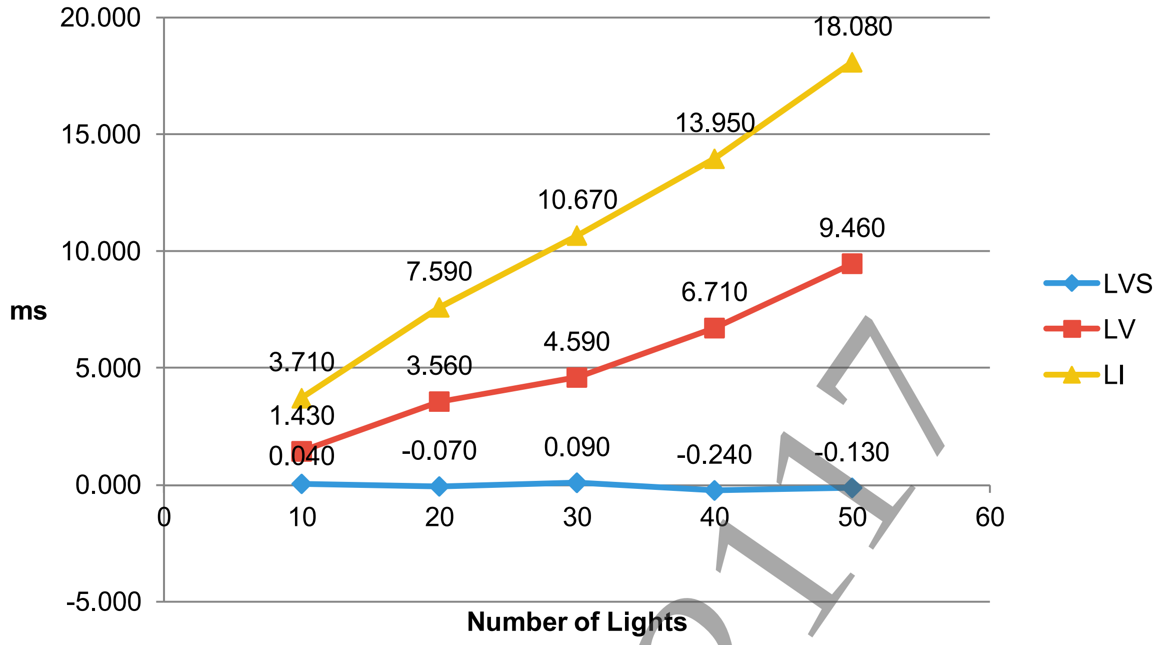
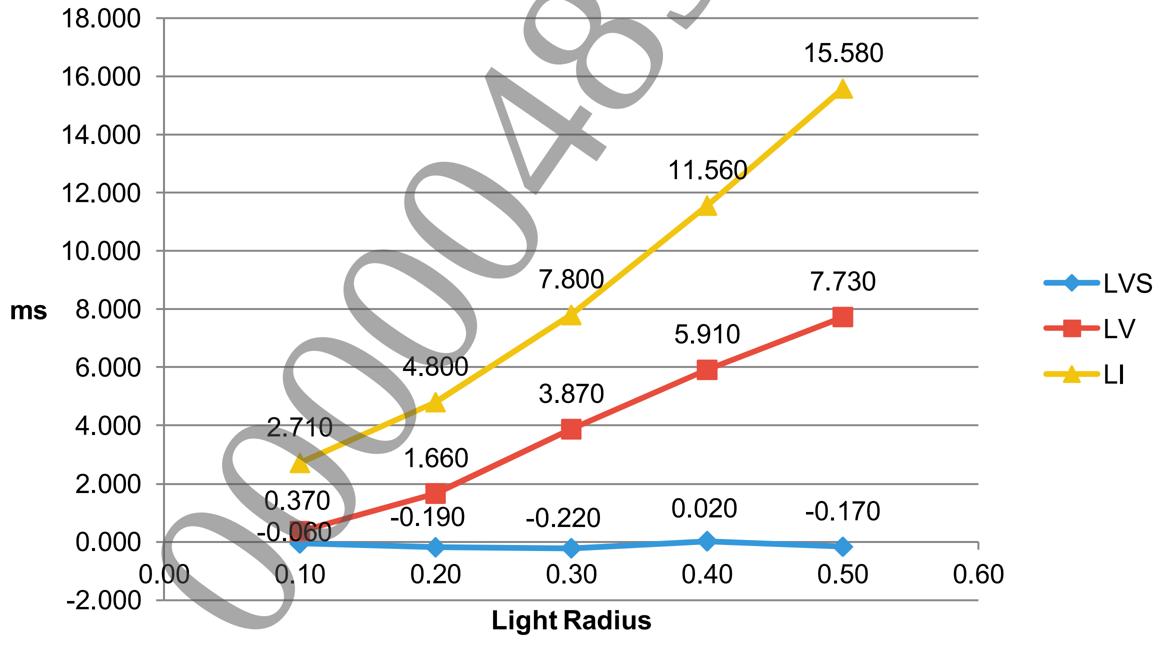


Figure 27 Light Prepass -
Difference in Lighting Pass Fragment Processing Time when Early Out is Enabled or Disabled



In the graphs, the lighting techniques shown are as follows:

- LVS: Light volume technique with the stencil test
- LV: Light volume technique without the stencil test
- LI: Indexed light technique with tile-based light culling

For the graphs with "Number of Lights" on the horizontal axis, the light radius is 0.45. For the graphs with "Light Radius" on the horizontal axis, the calculations were performed with 40 lights.

Full-Frame Performance

Figure 28 and Figure 29 are graphs of the time required for each lighting technique with the deferred rendering method as the number of lights and the radius of the lights changes. Figure 30 and Figure 31 are the analogous graphs for the light prepass method. Measurements were taken with all of the optimization techniques described in this chapter enabled.

It can be seen that, for both deferred shading and light prepass, the light volume technique using the stencil test offers the best performance. In contrast, the indexed light technique tends to be inferior to the techniques using light volumes. As explained in Chapter 3 "Lighting Techniques", the indexed light technique has the benefit that it need only reference the G-buffer once for each pixel. Nevertheless, one reason why it tends to be inferior is that, in the GPU of PlayStation®Vita, the overhead from accessing the G-buffer during the lighting pass is not a significant bottleneck, because the latency of memory reads for G-buffer accesses can be concealed as independent texture reads and sufficient bandwidth can be allocated by allocating the G-buffer to CDRAM. In this sample program, both deferred shading and light prepass perform the same shading; light prepass has poorer performance than deferred shading because light prepass renders the geometry twice, increasing vertex processing. The type of shading required must be considered when selecting between deferred shading and light prepass. If shading can be implemented using deferred shading, it is recommended that deferred shading rather than light prepass be used.

Figure 28 Deferred Shading - Frame Time

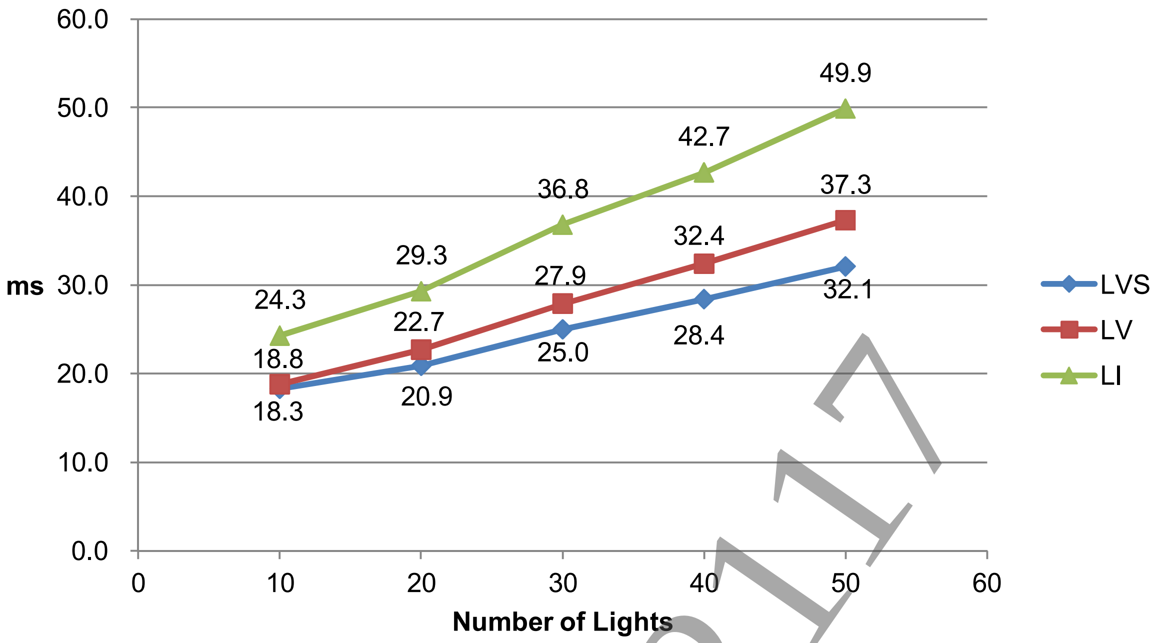


Figure 29 Deferred Shading - Frame Time

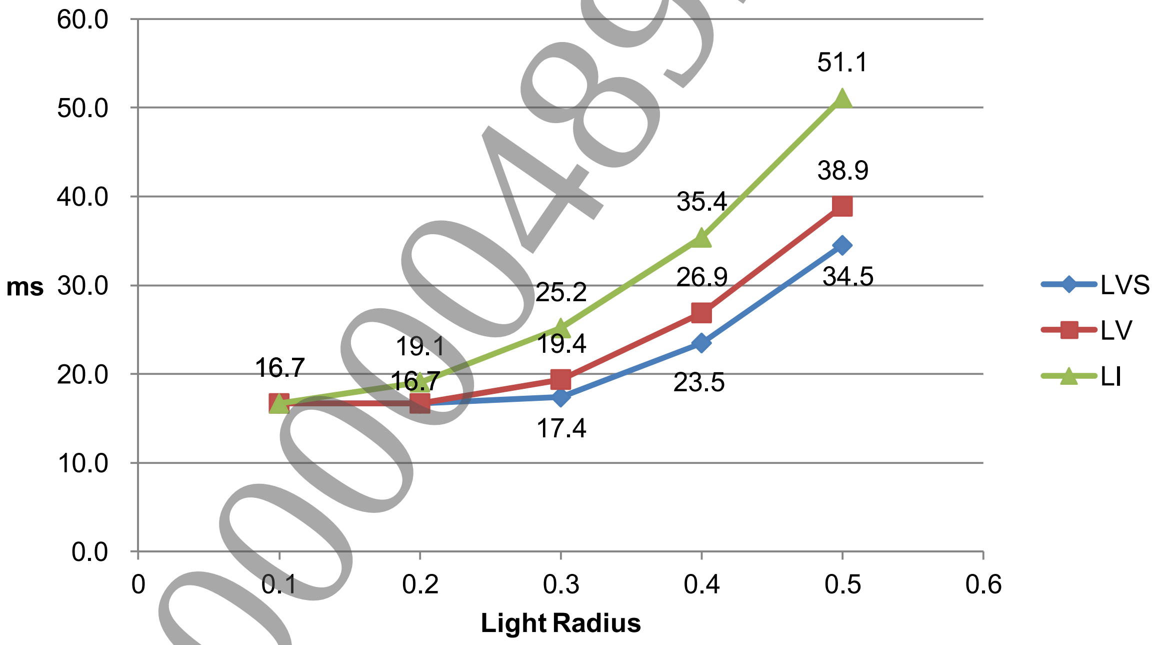


Figure 30 Light Prepass - Frame Time

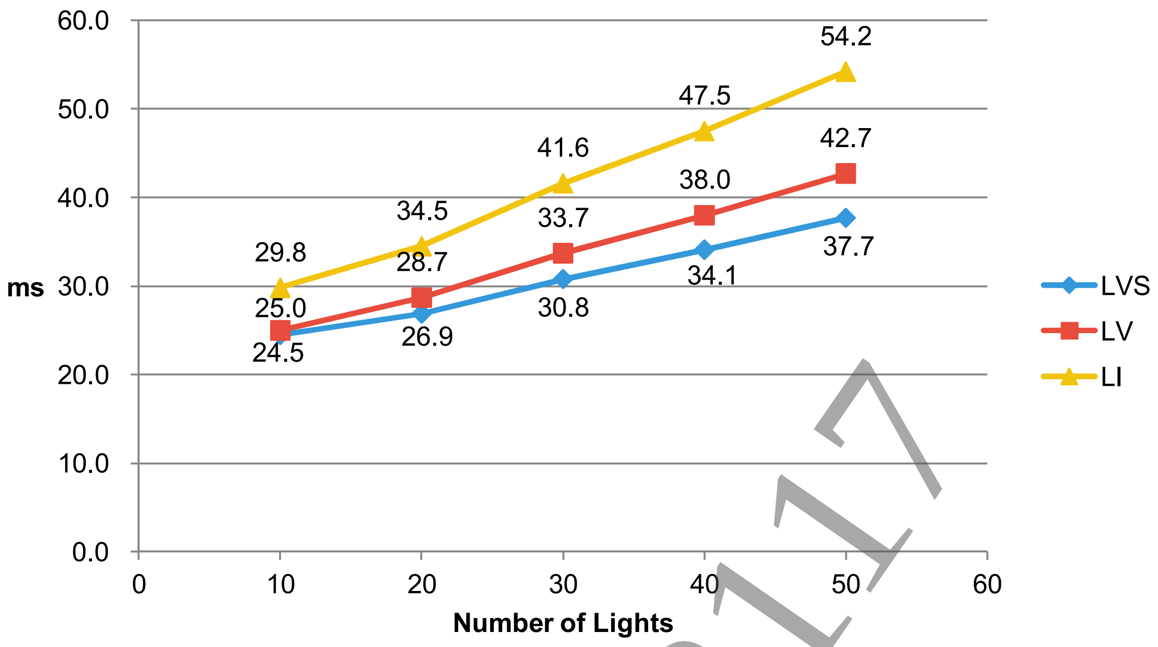
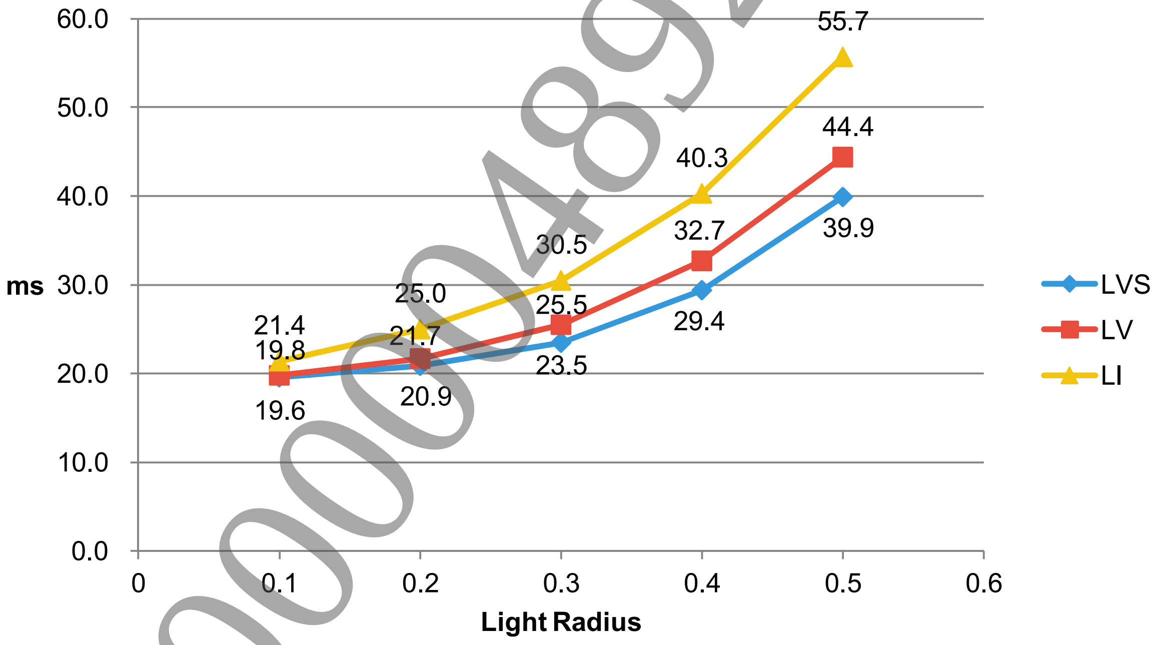


Figure 31 Light Prepass - Frame Time



In the graph, the lighting techniques shown are as follows:

- LVS: Light volume technique with the stencil test
- LV: Light volume technique without the stencil test
- LI: Indexed light technique with tile-based light culling

For graphs with "Number of Lights" on the horizontal axis, the light radius is 0.45. For graphs with "Light Radius" on the horizontal axis, the calculations are performed with 40 lights.