Precomputation Tutorial

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Document serial number: 000004892117

Table of Contents

About This Document

This tutorial demonstrates the use of libgxm's precomputation feature on PlayStation®Vita. The tutorial consists of this document and accompanying tutorial sample. This document describes details about how to create precomputed objects and how to set and update them to achieve CPU performance improvements. The objective is to precompute all or parts of the rendering API pipeline and to provide performance benchmarks based on the use of various precomputed or nonprecomputed states.

This document includes the following:

- Compares CPU costs for state setup using precomputed and nonprecomputed objects.
  - Breaks down the CPU overhead for vertex states, fragment states, and draw commands.
- Examines the trade-off between the number of objects for precomputation and the memory footprint (partial versus full precomputation).
- Shows the memory cost of using precomputed vertex states, fragment states, and draw objects.
  - Recommendation for efficient use of precomputed objects.

Related Documentation

Refer to the following related documents for additional details:

- *libgxm Overview*: Provides an overview of the libgxm graphics library and how function calls interact with the underlying GPU.
- *libgxm Reference*: Provides detailed reference descriptions of each of the libgxm API functions and data structures.
- *Performance Analysis and GPU Debugging*: Provides detailed descriptions of Razor for PlayStation®Vita (Razor) GPU performance analysis and debugging features.

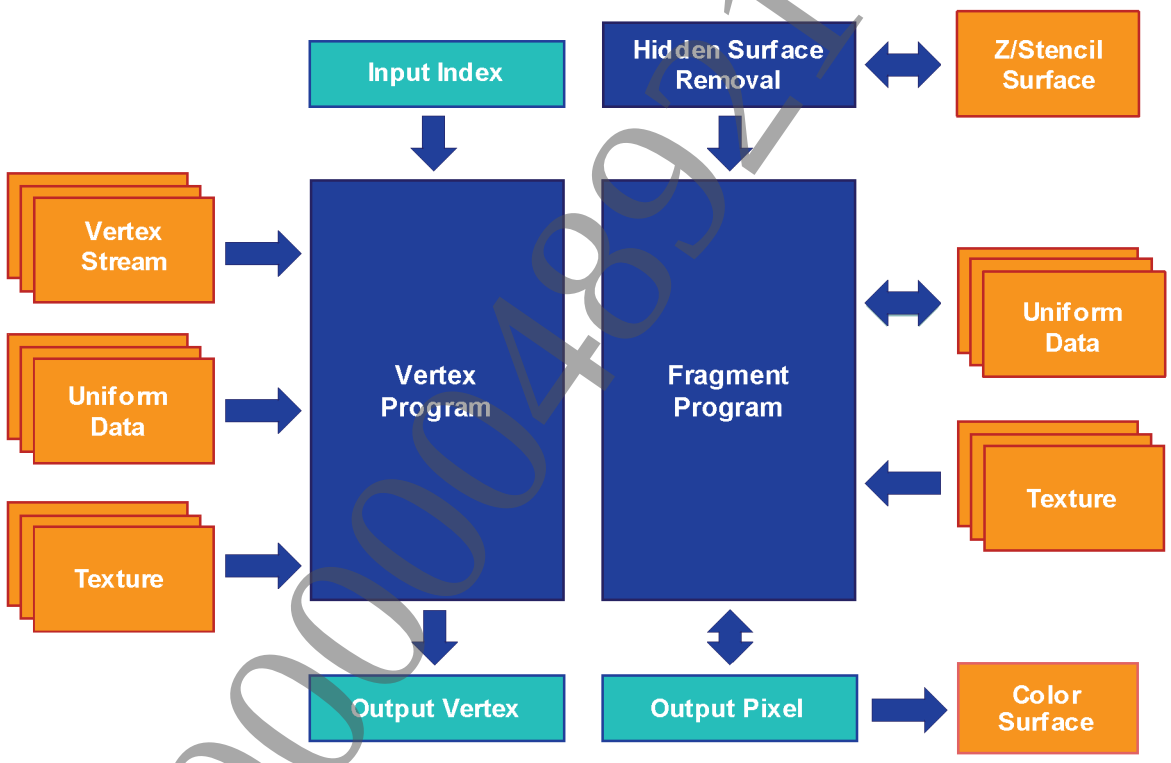
1 Introduction

The libgxm rendering API provides a set of highly optimized calls that are designed for efficient rendering. It constructs a command stream that sets up the state data for vertex and fragment processing and draws the indexed primitives. An important libgxm feature is that it minimizes the number of state changes, which reduces the need for flushing the graphics pipeline. It provides a rendering context with a persistent state; that is, if a state is set, it stays set indefinitely within the scenes for all draw calls that are submitted to the GPU firmware layer for rendering.

Rendering Pipeline

The libgxm rendering API efficiently maps to the SGX architecture and facilitates easy configuration of its hardware units for both vertex- and fragment-processing pipelines. It achieves this by generating a hierarchy of internal data structures; this hierarchy allows states to be flushed and sets up the components required for the various stages of the rendering pipeline. Figure 1 shows the components that are supported by libgxm and how they are connected to the vertex and fragment shader units.

Figure 1 libgxm Rendering Pipeline Showing Items Set by libgxm



Depending on which libgxm API calls are issued, the various rendering pipeline components get set. The setup involves regenerating the data structure hierarchy for the different categories of the rendering pipeline. These categories are:

- **State setup:** If render states change between draw calls, libgxm must generate a state delta for the GPU. This involves creating a small buffer with the changed state and a small program to emit it to the GPU. Most state data is consumed directly by the vertex-processing pipeline, but some (such as depth/stencil mode) is serialized into the parameter buffer for use by the fragment pipeline during the fragment-processing phase.
- **Vertex pipeline data:** If the vertex program, vertex textures, or vertex uniform buffers change between draw calls, libgxm may need to regenerate some internal buffers and programs for the GPU, depending on what is referenced by the current vertex program.

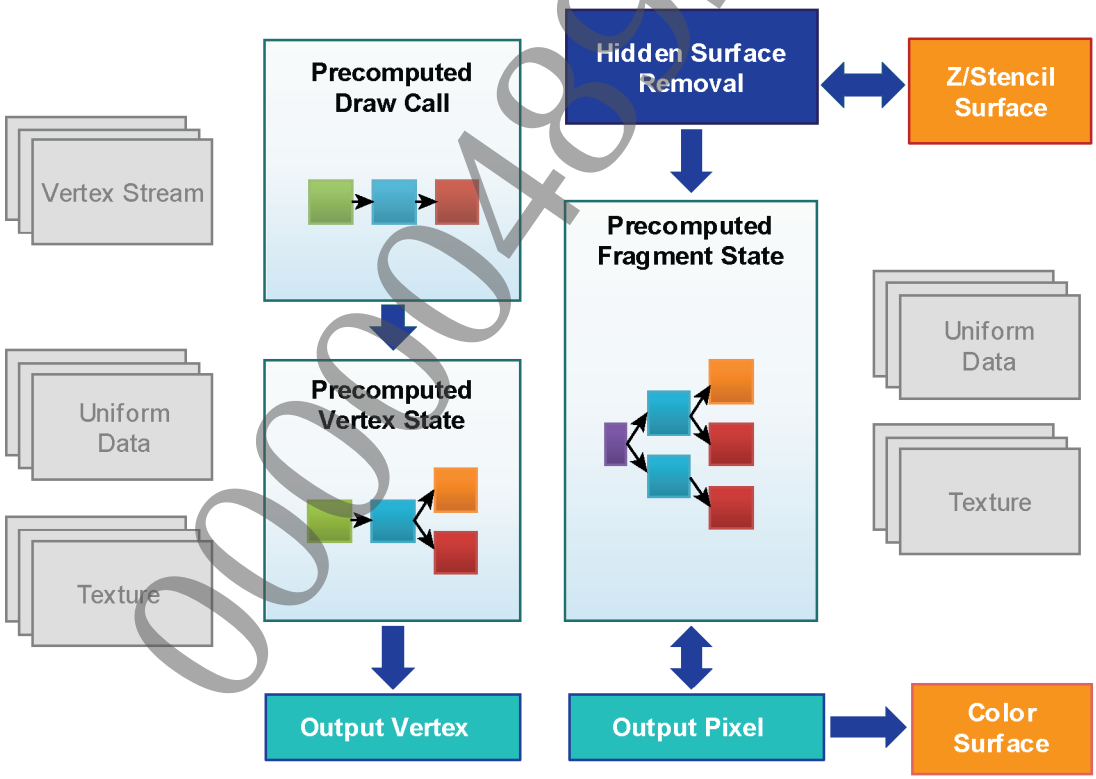
- **Fragment pipeline data:** Similar to the vertex pipeline, if the fragment program, fragment textures, or fragment uniform buffers change between draw calls, libgxm may need to regenerate some internal buffers and programs for the GPU. The top level of this pipeline handles tiling unit state update, with state delta that includes the state that changes as the fragment program is changed. The update will associate the triangles in future draw calls with the programs and data buffers in the parameter buffer. This information is used by the fragment pipeline during the fragment-processing phase.
- **Draw Commands:** If the vertex program or vertex streams change between draw calls, libgxm needs to regenerate the VDM draw command to issue the draw calls which encodes the updated draw parameters and index buffer address as well as holds a pointer to the changed vertex program that is run for each vertex.

Storage for these dynamically generated programs and buffers is reserved in the libgxm context ring buffers. Unfortunately, CPU overhead occurs as a result of tracking and flushing various states between draw calls and generating the data structures to set buffers and programs that the GPU needs to render. This overhead can be avoided by precomputing these data structures.

Precomputation

The precomputation feature of libgxm gives you the option to precompute data structures in memory for the part of the rendering pipeline. Using precomputed objects avoids having to set many types of states on the libgxm context on the fly, with states patched in the precomputed object itself (Figure 2). This results in more efficient CPU usage, though at the cost of more memory usage.

Figure 2 The Updated Pipeline When Using Fully Precomputed Data Structures



Different parts of the pipeline can be precomputed independently, giving the flexibility of precomputing only some structures while setting up others using libgxm context, thus minimizing the memory cost. Additionally, some of the pipeline cannot be precomputed, so you still need to configure it using the libgxm context calls. This includes setting up the vertex and fragment programs for which the precomputed objects are created and the render state data, such as cull mode, depth modes, and stencil states.

Precomputed Objects

As shown in Figure 2, the API implements three types of precomputed objects: fragment state, vertex state, and draw calls, each based on the internal data dependencies discussed in the previous section. These precomputed objects are discussed in this section.

Precomputed Vertex States

The precomputed vertex state bypasses the need for libgxm context calls to set and reserve uniform buffers and to set vertex textures, using instead the uniform buffers and textures set on the precomputed vertex state. You must provide some GPU memory for the precomputed vertex state for the given vertex program by calling the function `sceGxmPrecomputedVertexStateInit()`. The memory is used directly for the various internal buffers and programs required by the GPU for this vertex program.

You can query the amount of memory used for a precomputed vertex state by calling `sceGxmGetPrecomputedVertexStateSize()`.

To embed these buffers and textures into a precomputed vertex state, replace the specific context calls with the corresponding precomputation calls as shown in Table 1.

Table 1 libgxm Context Calls Replaced When Using a Precomputed Vertex State

libgxm Context	Precomputed Object
`sceGxmSetVertexUniformBuffer()`	`sceGxmPrecomputedVertexStateSetAllUniformBuffers()`, `sceGxmPrecomputedVertexStateSetUniformBuffer()`
`sceGxmSetVertexTexture()`	`sceGxmPrecomputedVertexStateSetTexture()`, `sceGxmPrecomputedVertexStateSetAllTextures()`
`sceGxmSetVertexDefaultUniformBuffer()`, `sceGxmReserveVertexDefaultUniformBuffer()`	`sceGxmPrecomputedVertexStateSetDefaultUniformBuffer()`

Precomputed Fragment States

Similarly, if using precomputed fragment state, it is possible to bypass setting and reserving fragment uniform buffers and setting fragment textures on the libgxm context, using instead the uniform buffers and textures set on the precomputed fragment state. You must provide some GPU memory for precomputed fragment state for the given fragment program by calling the function `sceGxmPrecomputedFragmentStateInit()`. The memory will be used directly for the various internal buffers and programs required by the GPU for this fragment program.

You can query the amount of memory used for a precomputed fragment state by calling `sceGxmGetPrecomputedFragmentStateSize()`.

To embed this information into a precomputed fragment state, replace the specific context calls with the corresponding precomputation calls as shown in Table 2.

Table 2 libgxm Context Calls Replaced When Using a Precomputed Fragment State

libgxm Context	Precomputed Object
`sceGxmSetFragmentUniformBuffer()`	`sceGxmPrecomputedFragmentStateSetAllUniformBuffers()`, `sceGxmPrecomputedFragmentStateSetUniformBuffer()`
`sceGxmSetFragmentTexture()`	`sceGxmPrecomputedFragmentStateSetTexture()`, `sceGxmPrecomputedFragmentStateSetAllTextures()`
`sceGxmSetFragmentDefaultUniformBuffer()`, `sceGxmReserveFragmentDefaultUniformBuffer()`	`sceGxmPrecomputedFragmentStateSetDefaultUniformBuffer()`

Precomputed Draw Commands

Finally, if you are using precomputed draw, it is possible to replace any vertex streams set on the libgxm context with the vertex streams set on the precomputed draw. You must provide some GPU memory for an internal program required by the GPU for the given vertex program by calling the function `sceGxmPrecomputedDrawInit()`.

You can query the amount of memory used for a precomputed draw object by calling `sceGxmGetPrecomputedDrawSize()`.

To embed this information into a precomputed draw command, replace the specific context calls by corresponding precomputation calls as shown in Table 3.

Table 3 libgxm Context Calls Replaced When Using a Precomputed Draw Call

libgxm Context	Precomputed Object
`sceGxmDraw()`, `sceGxmDrawInstanced()`	`sceGxmPrecomputedDrawSetParams()`, `sceGxmPrecomputedDrawSetParamsInstanced()`
`sceGxmSetVertexStream()`	`sceGxmPrecomputedDrawSetVertexStream()`, `sceGxmPrecomputedDrawSetAllVertexStreams()`

Patching and Synchronization

To correctly use the precomputed structures after they are generated, patch them dynamically with any information that has changed – such as default uniform buffers or textures – for each frame. Because the precomputed structures do not have any dependencies, they can be patched at any time. However, be careful that the object being patched is not being used by the GPU, because otherwise the CPU will overwrite the used data.

Therefore, for safe patching it is essential to synchronize the CPU writes with the GPU usage. To do this, use the notification mechanism provided by libgxm. One notification is written to memory at either end of the vertex pipeline, at which point all the precomputed vertex states or dynamic draw data are safe to patch. The other notification is written at the end of the fragment pipeline, at which point it is safe to patch any dynamic precomputed fragment states.

Figure 3 Safe Patching by Synchronizing CPU Writes and GPU Usage Using Single and Double Buffers

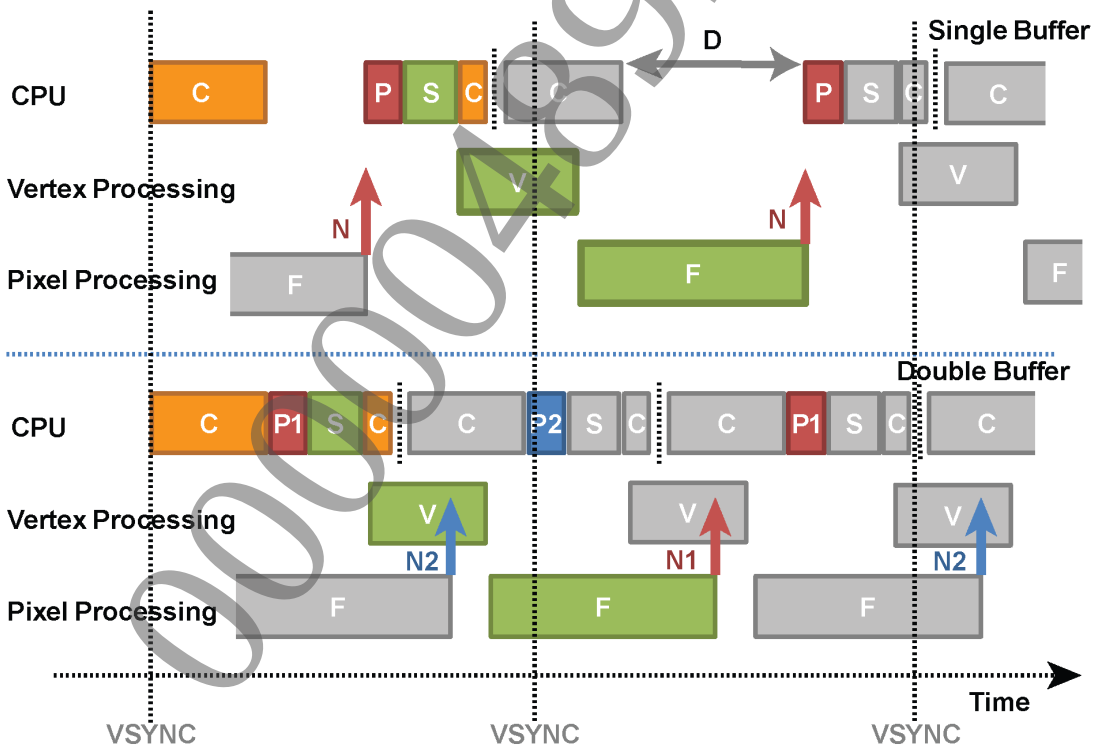


Figure 3 shows a scenario with dynamic precomputed fragment state. The top part shows usage with a single buffer and the bottom shows a more efficient usage with a double buffer of precomputed fragment states. The blocks in the figure represent the following jobs:

C – CPU processing

P – Precomputed fragment state patching to single buffer by CPU

S – Scene built by CPU and submitted to the firmware

V – Vertex processing for the scene

F – Fragment processing for the scene

N – Notification of fragment pipeline completed when using single buffer, safe patching job ***P*** can start

P1, P2 – Precomputed fragment state patching to buffers 1 and 2 by CPU, when using a double buffer

N1, N2 – Notification of fragment processing completed using precomputed objects in buffer 1 and 2, when using a double buffer

Precomputation trades improved CPU performance for memory usage, so you need to consider memory usage for the precomputed objects. If you need to minimize memory usage, you can precompute only parts of the objects. For example, you could precompute only parts of the hierarchy, mainly the structures that give the most benefit in CPU usage, and use libgxm context calls for the rest of the hierarchy. Note that if you use a double or triple buffer for synchronization, the result will be a doubling or tripling of the memory that your dynamic precomputed objects use.

Benefits of Precomputation

Using precomputed objects not only reduces CPU overhead by reducing state tracking and the number of flushes, but also provides other useful options. Because each of the precomputed structures is self-contained with no dependencies, you can update these on different threads when the structure is not being used by the GPU.

Thus, in contrast to using the API without precomputation, which requires that rendering must be performed from a single thread, state changes for the precomputed objects can be patched over multiple independent threads in parallel, resulting in CPU cost benefits.

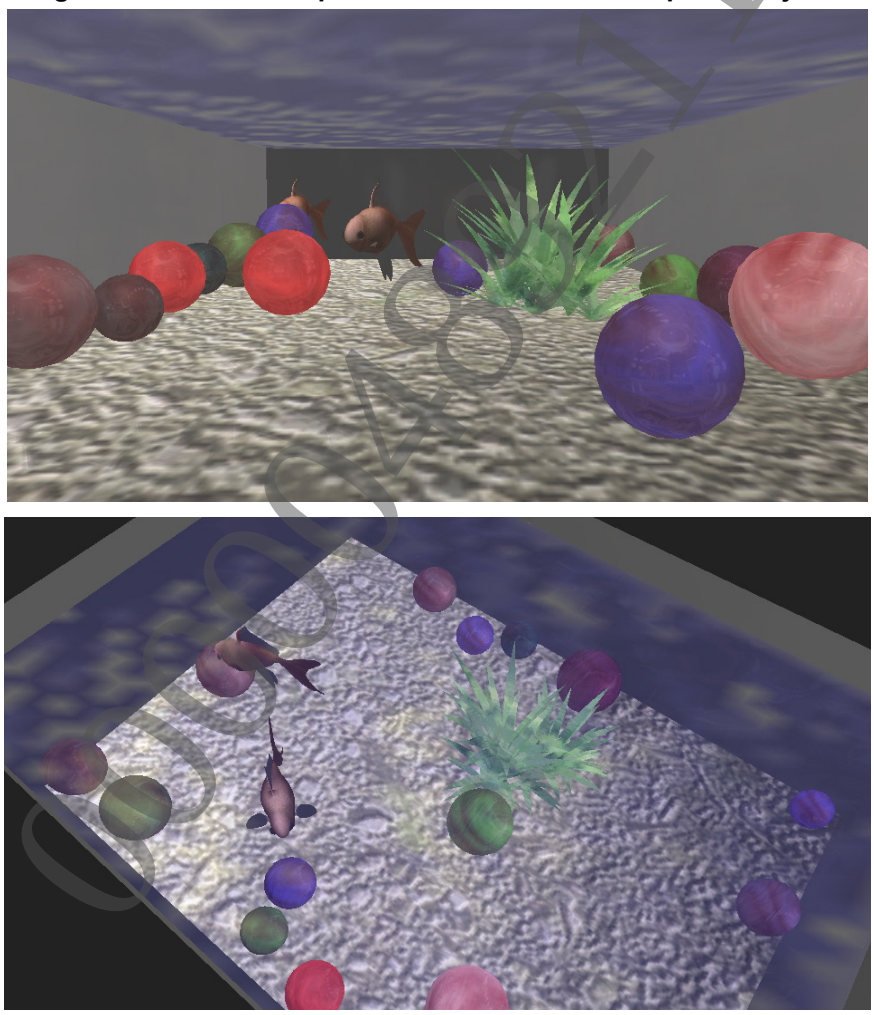
2 Tutorial Walkthrough

This chapter describes the tutorial, which tests the performance of libgxm's precomputation feature on PlayStation®Vita. The software shows how to precompute different states and draw objects for a draw call and use them for improving performance. The software also uses objects that are not precomputed so that we can compare the performance between the two pipelines. The additional memory usage when using precomputed objects is also compared, with the option of using a partial set of precomputed objects to balance performance improvement with optimal memory usage.

Precomputed Scene

The software implements and displays an aquarium with a mixture of precomputed and nonprecomputed objects. The scene renders multiple objects, including the water surface, an animated fish, a sea plant, and multiple instances of spherical playballs. A caustics effect is applied to the aquarium floor.

Figure 4 Rendered Aquarium Scene with Precomputed Objects



The software generates precomputed objects for the fish, a static plant, and the ball instances, and can combine them with nonprecomputed states for each of the draw calls. By default, the scene is rendered using the nonprecomputed structure for the fish, sea plant, and the playballs.

The software also has a synchronization mechanism and demonstrates how to use multiple buffers for patching the dynamic precomputed objects.

Skinned Fish Draw Calls

The fish is a COLLADA mesh that contains skinning data and has three controller children, each with one geometry rendered as a separate draw call.

The fish geometries are animated using a skinning vertex shader that has a higher complexity due to the need for more skinning-related uniform buffers, including user-defined uniform buffer for skinning matrices. This shader includes instructions to perform animation processing and calculate the final position for each of the vertices. This requires blending the contributions from several matrices with the amount of influence set by their corresponding bone weights. The vertex shader expects:

- Skinning matrices that are set as uniform buffers to the shader by libgxm
- Blend indices to fetch the relevant matrices and blend weights stored per vertex
- Bind shape matrix that transforms the bind-shape from the object-space to bind-space thus binding it to the fish skeleton, passed in as uniform parameters for the draw call

The fragment shader for the fish performs per-pixel Phong shading. For the effect, libgxm needs to set up:

- Diffuse and normal map textures
- Uniform parameters for calculating diffuse, specular, and ambient lighting contributions

For the fish, the software precomputes the following:

- The draw commands corresponding to each of the three draw calls, with base addresses of the particular vertex streams embedded along with other parameters to draw the fish geometry (such as the pointer to index data and the number of indices).
- The vertex states that store data structures required for processing the mesh vertices for each of the fish instances that includes structure to read skinning matrices from memory, and other vertex program uniforms. The default uniform buffer that includes the bind shape matrix is allocated and assigned to the corresponding vertex state.
- The fragment states that store data structures required for the Phong lighting calculations and texture mapping for each fragment. The default uniform buffer to store the light color, light position, eye position, and other lighting calculation parameters is allocated and assigned to this precomputed state. Also the associated texture states for diffuse and normal maps are set for the draw calls.

Sea Plant Draw Calls

The sea plant is another COLLADA model loaded in the scene. This is a simple mesh with a single static geometry.

The plant geometry is rendered using a simple vertex shader that performs per-vertex lighting calculations. The vertex shader expects:

- Uniform parameters for performing transformation and diffuse, specular, and ambient lighting contributions for the draw call.

The fragment shader for the plant performs simple texture mapping and requires setting up the diffuse texture states.

For the plant, the software precomputes the following:

- Draw calls that embed the draw commands to store base addresses of the vertex streams for the plant mesh along with, the information on draw parameters – such as the types of primitives to be rendered, the types of index data, and pointers to index data and number of indices.
- The vertex states for the corresponding vertex program for lighting calculations. The default uniform buffer to store transformation and lighting parameters is allocated and assigned to the corresponding vertex state.
- The fragment states for the fragment program to perform standard texture mapping. The diffuse texture map is set for the plant draw call.

Playball Draw Calls

Next, the software renders multiple instances of a simple textured sphere with environment mapping. The associated geometry for all instances is the same.

The spheres are rendered using a simple vertex shader that performs local to world transformation, calculates the lighting contribution, and composites the lighting with the geometry color for each of the instances. These spheres are attached to the collision simulator, so the world matrix for each instance needs to be updated and passed to the vertex shader. The vertex shader expects:

- Uniform parameters for performing transformation and diffuse, specular, and ambient lighting contributions for each of the sphere instances.

The fragment shader for the plant performs simple texture mapping and reflective environment mapping. It requires setting up:

- The diffuse and cubemap texture states

Similarly to the skinned fish and the sea plant, the tutorial precomputes the draw, vertex, and fragment states for all the playball instances. The precomputed objects include the following:

- One precomputed draw call that stores base addresses of the vertex streams for the sphere mesh. Also, draw parameters such as the type of primitives to be rendered, the type of index data, and pointers to index data and number of indices are embedded into the object.
- The vertex states that store transform and lighting parameters corresponding to each instance, with world matrix, light, and view positions patched to the embedded default uniform buffer at runtime assigned to the corresponding vertex state.
- The fragment states for the fragment program that performs standard texture mapping. The diffuse texture map and cube map are set for the playballs for the corresponding state. A fragment state is computed for each variation (different texture/color) of the playball to be rendered.

Aquarium

The scene also renders an aquarium that includes the four boundaries, the floor, and the animated water surface. The boundaries and the floor are added as fixed rigid bodies into the collision simulator.

The boundaries are rendered as simple transparent planes. The water surface is a tessellated mesh with its vertices animated using a simple sinusoidal wave. The rendering includes environment mapping to simulate reflection from the water surface. The aquarium floor is a textured plane with normal and specular mapping applied to highlight marbles. In addition, a realistic underwater caustics effect is added to the floor using the water surface normals and depth of the pool.

Note that these draw calls are rendered using the standard nonprecomputed pipeline.

3 Performance Analysis

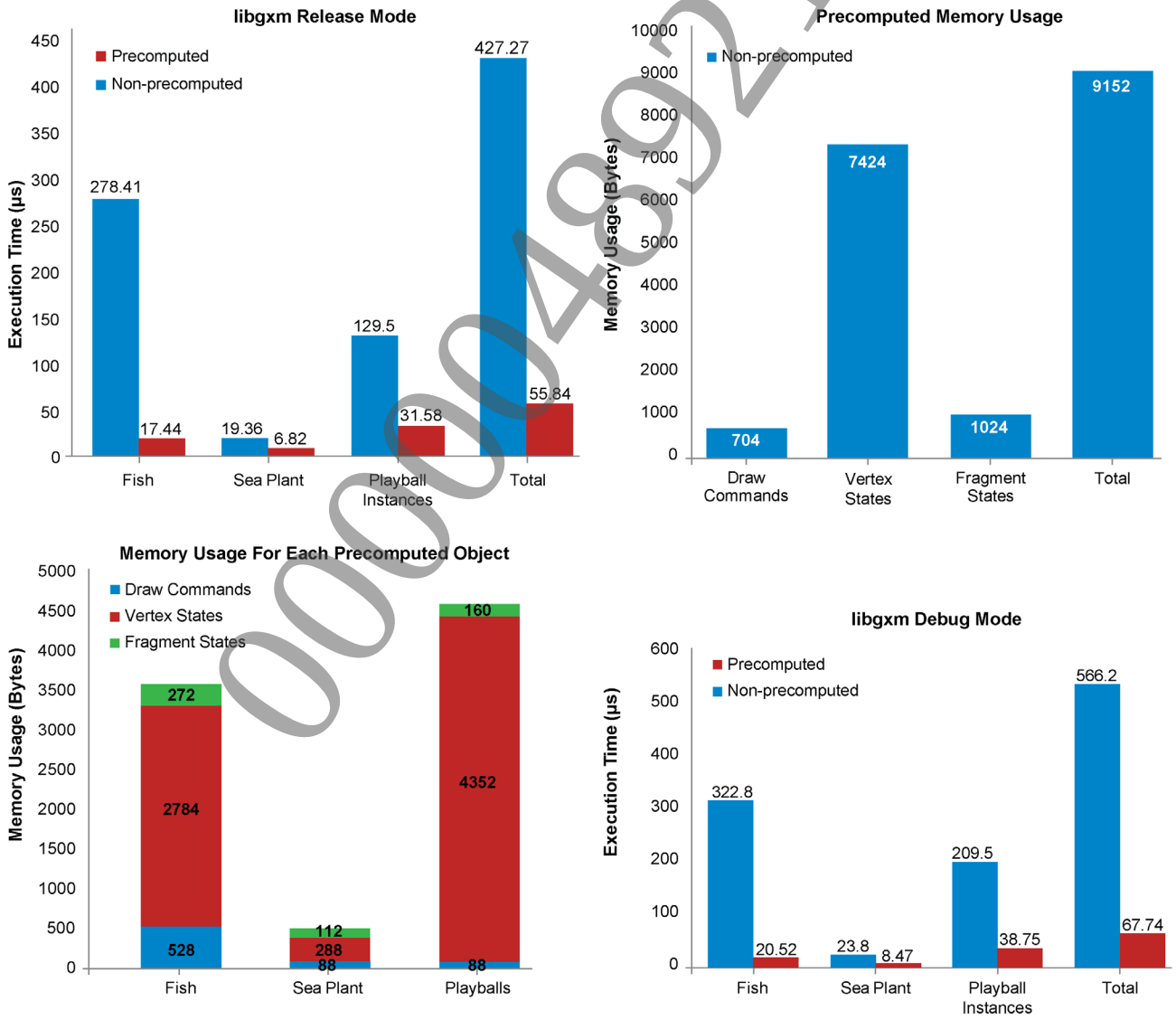
This chapter describes some of the measurements made for the precomputation tutorial sample application described in the previous chapter. The performance analysis is based on the measurements obtained by performing CPU traces with Razor. For each object to be rendered, user markers are set around the code using libgxm API calls to set up the states. Traces are then captured for the standard nonprecomputed pipeline, for the precomputed states, and for partially precomputed objects for each fish, sea plant, and play ball.

Precomputed versus Nonprecomputed Performance

The first measurement is to compare the performance of a nonprecomputed pipeline with a fully precomputed one. This is done based on the libgxm cost for setting up the states for both cases. The cost is given by the timing of the user markers set for the CPU traces in Razor.

Figure 5a shows the measurements for the two cases, which makes it clear that using precomputed objects drastically reduces the libgxm cost of setting up the states.

Figure 5 Performance Comparison When Using Nonprecomputed and Precomputed Structures (Fig. 5a top left, 5b top right, 5c lower left, 5d lower right)



However, as shown in Figure 5b, using precomputed objects requires memory allocation. The measurements indicate that precomputed vertex states have the most memory allocations. Considering all geometries separately, the precomputed objects for the fish and playball instances consume much more memory in comparison to the precomputed sea plant objects as indicated in Figure 5c.

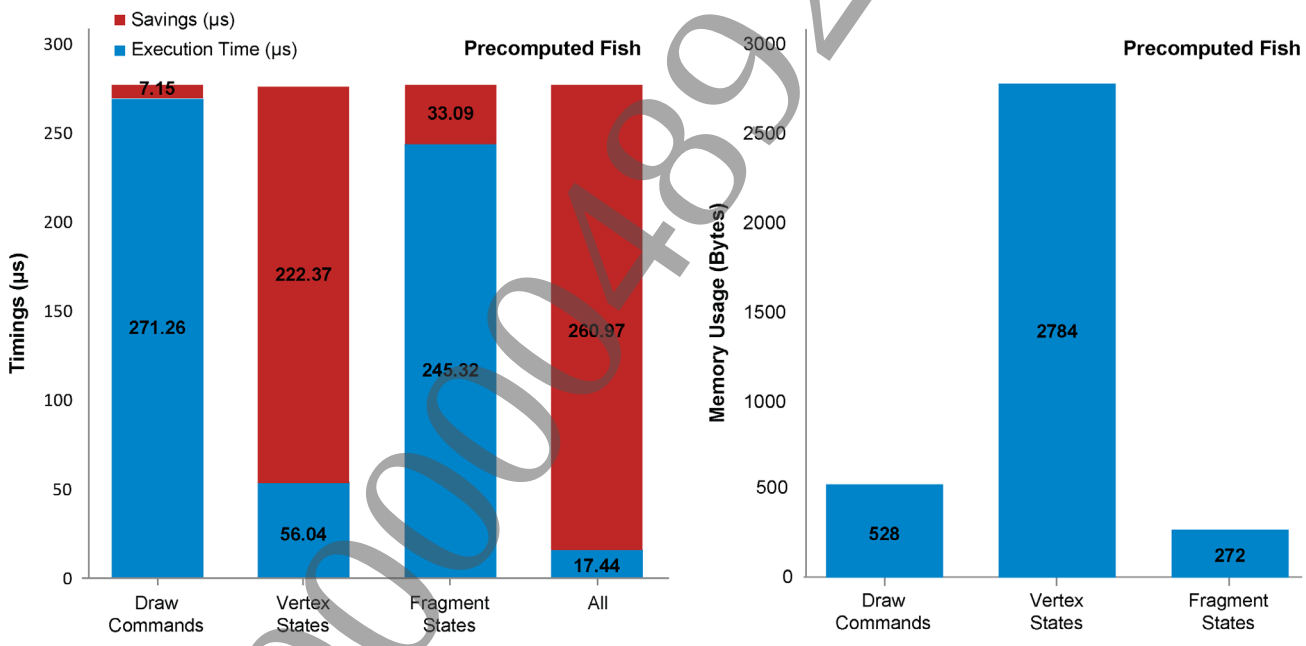
Also for comparison, Figure 5d shows the test results with the debug version of libgxm. The debug version employs additional state validation for the precomputation functions and so, as expected, savings are lower than in the release version.

Next we analyze the performance for each of the objects rendered in the scene using both nonprecomputed and precomputed structures. Note that all the following measurements are made for the release version of libgxm.

Precomputed Fish

For fish geometry, there are three associated draw calls for rendering, with each having its own set of skinning data. A single instance of the fish uses three precomputed draw and vertex state objects, and one single precomputed fragment state object. The measurements show that using precomputed objects saves CPU time on each of the precomputed objects, though the biggest savings is achieved for the precomputed vertex states as shown in Figure 6a, which shows the individual timings and savings when using the three precomputed structures for the fish one at a time.

Figure 6 Savings and Memory Usage for Precomputed Fish Objects (Fig. 6a left, 6b right)



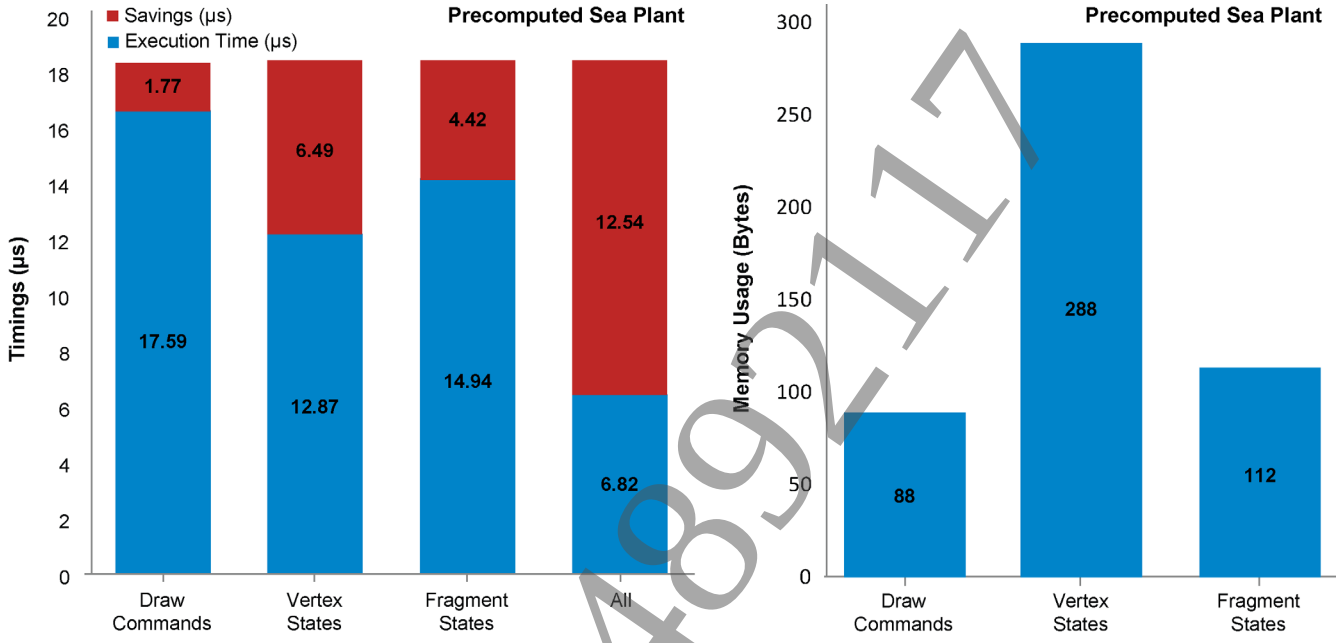
The last column in Figure 6a is the total time and savings for a fully precomputed fish that is equal to the sum of individual savings for each object, which is expected, as all the objects are independent and share no data. Figure 6b shows the memory usage for each of the precomputed objects. The vertex state requires considerably more memory than the draw calls and the fragment state objects. This can be attributed primarily to the complexity of the vertex program that requires setting up a large default uniform buffer and a uniform buffer for the skinning matrices, and additionally requires three precomputed objects, one for each draw call corresponding to the fish geometry.

On analyzing, for a trade-off you can use precomputed vertex states for high CPU cost benefits and use non-precomputed structures for the other parts of the fish to save memory.

Precomputed Sea plant

The sea plant geometry has one each of precomputed draw, vertex state, and fragment state objects. As shown in Figure 7(a), the measurements on the cost of flushing the state on any of the hierarchies using precomputed objects saves CPU cost for each of the precomputed objects. Unlike the skinned fish, the sea plant has static precomputed states and precomputed draw, which is beneficial because not patching the data results in relatively higher CPU time savings. Additionally, the memory usage for each of the precomputed objects is quite low (Figure 7b) and total usage is much lower in comparison to other objects, as seen in Figure 5c.

Figure 7 Savings and Memory Usage for Precomputed Sea Plant Objects (Fig. 7a left, 7b right)

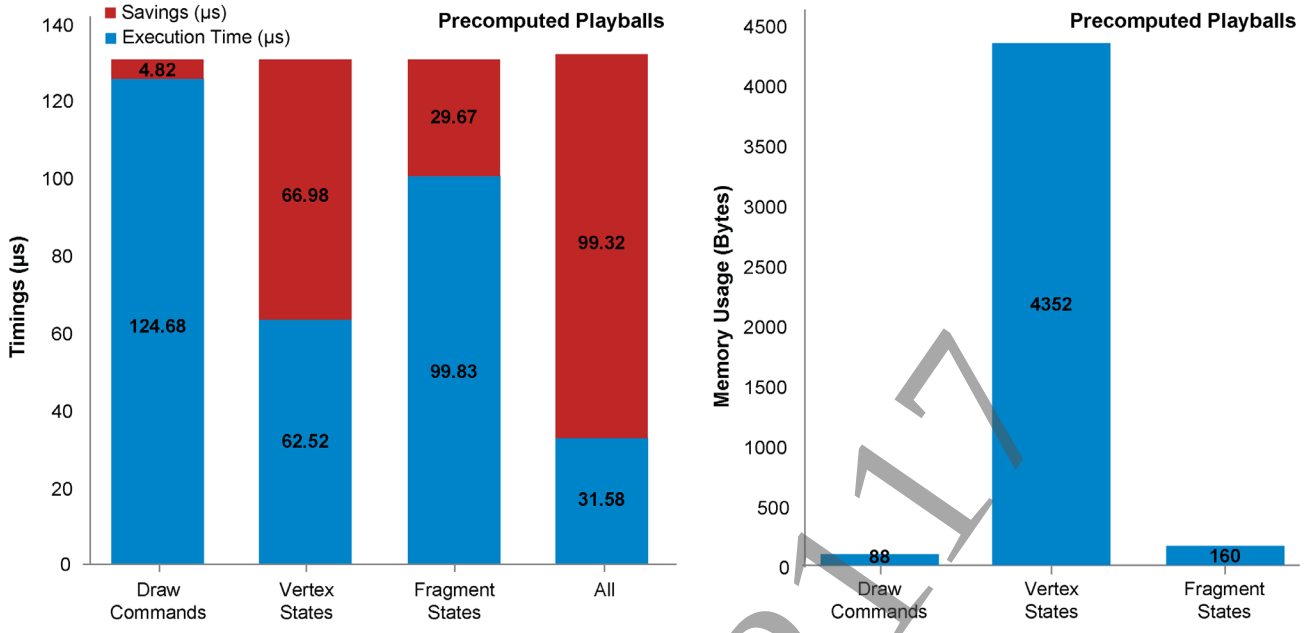


Because using either of the precomputed objects for the sea plant does not provide significant CPU savings relative to other objects, you can just use the standard non-precomputed pipeline for rendering it.

Precomputed Playballs

In the precomputed playballs scene, the tutorial sample renders 16 playball instances with one precomputed draw call, using 16 different vertex states with varying world transformation matrices and color, and one fragment state object. Because this sample uses precomputed objects for each state, a large amount of CPU time is saved, specifically for the vertex and fragment states (Figure 8a). The high savings for the vertex state can be attributed to the multiple instances, each with its own independent vertex state that is patched with the uniform data, avoiding dynamic state updates by libgxm. The savings from using a precomputed fragment state are a result of bypassing dynamically setting up the diffuse and cubemap texture setup, which are embedded and used for computing the final composited fragment color.

Figure 8 Savings and Memory Usage for Precomputed Playball Instances (Fig. 8a left, 8b right)



The memory usage for each of the precomputed objects is quite low for the draw call and the fragment state but quite high for the vertex state, mainly because of multiple instances needing multiple precomputed structures (Figure 8b).

For the playballs, using precomputed vertex states for all the instances provides a good CPU cost benefit but with very high memory usage, while precomputed fragment state – although it uses low memory – gives a relatively good CPU cost savings. The precomputed draw call in this case does not provide significant CPU savings and so you could draw this using a normal pipeline.

Recommendations

Based on the performance analysis:

- As expected, using precomputed objects saves on the CPU cost of setting states. Also, using libgxm's release version gives better saving compared to libgxm's debug version and we expect performance to be even better for other complex scenarios. It is thus recommended that you use the release version of libgxm when you want to analyze performance.
- Shader complexity can affect the cost of flush between the draw calls. So shaders that reference a lot of resources like uniform buffer or textures will result in larger and complex programs and data buffers. Precomputing such objects can result in CPU performance improvement, though at a cost of higher memory usage. Additionally, the instance and variation count of each of the geometries affects the performance/memory tradeoff.
- When using precomputation, it is recommended that you analyze the CPU savings that it provides relative to any increase in memory usage. This will help you to precompute objects only when it is necessary and effective. You do not need to precompute all the objects; precompute those objects where there is a balance between CPU cost savings and extra memory usage.
- The use of static precomputed state and precomputed draw should be favored because not patching data dynamically will get you the best results in terms of maximizing CPU time savings.
- The CPU savings from each precomputed object adds up to give the total savings. This is expected, as the various precomputed objects do not have data dependencies, which also facilitate the use of partially precomputed objects and independent patching of updated objects on separate threads, and can be exploited to achieve better performance.

4 Implementation

This chapter describes the implementation details for the aquarium tutorial sample application.

The tutorial sample is located at:

```
SDK/target/samples/sample_code/graphics/tutorial_precomputation
```

The tutorial sample renders the aquarium scene with setups for both precomputed and nonprecomputed objects with the option of dynamically switching from one to another for each object using the on-screen debug menu.

Scene Initialization

The aquarium scene involves the initialization of resources for a number of geometries, which includes textures, data, and shaders for different scene effects. The function calls for performing these initializations are included in `resource_util.cpp`. The application does the complete initialization by calling:

```
// set up collada mesh data, textures and shaders for objects in the scene
g_sceneResources.initialize(m_graphicsData.pShaderPatcher);
```

The implementation also involves collision simulation between different geometries in the scene. This is done by creating a simulator and attaching different geometries to the simulator as rigid bodies, the implementation of which is in `rigid_body.cpp`.

```
// pass the bounds of the aquarium, maximum number of objects for collision checks,
number of large rigid bodies, and gravity coefficient for the scene
m_simulator.initialize(…);
```

The aquarium bounds, the floor, the water surface, and the playballs are simple planes and spheres that are created and initialized for the scene by calling:

```
m_waterSurface.initialise(); // creates a tesselated mesh for given number
                             of sections

m_aquarium.initialise(m_simulator); //creates aquarium bounds and floor
                                    geometries (Plane) with corresponding
                                    rigid bodies (fixed) and collision enabled.

m_playBalls.initialise(m_simulator); //creates a sphere geometry used for all
                                     the playball instances. Rigid body is set
                                     for each instance and collision is
                                     enabled.
```

These functions are implemented in `scene.cpp`.

Creating Precomputed Objects

As part of the aquarium scene, the tutorial sample also needs to initialize the precomputed structures for the fish mesh, the sea-plant mesh, and the playball instances. This is done by calling:

```
// pass the shader effect pointer and shader parameters as required
// for each object to be precomputed
m_fishes.initialisePrecomputedData(effect::SkinningEffect *shaderEffect, ...);
m_plants[0].initialisePrecomputedData(…);
m_playBalls.initialisePrecomputation(effect::TextureEffect);
```

This function mainly uses the specific shaders associated with the geometries to compute the precomputed draw, vertex, and fragment states. The precomputation-related functions are included in `precomp_util.cpp`, which includes initialization and drawing of precomputed objects. The precomputed structure for each geometry is initialized by calling:

```
precomputation::PrecompData *pPrecompData = new precomputation::PrecompData;
pPrecompData->initialize(
    SceGxmVertexProgram *vtxProg,    // the patched vertex program
    SceGxmFragmentProgram *fragProg, // the patched fragment program
    uint32_t numGeoms, // number of associated draw calls with the geometry
    uint32_t numInstances, // number of different vertex states to be created
                           (based on number of draw calls for the geometry and
                            number of its instances for the scene)
    uint32_t numVariations) // number of different fragment states to be created
                           (based on number of variations in color and textures
                            to be used);
```

Depending on the type of the object to be created (draw/vertex/fragment), a specific function with specific arguments and the object index is called. To create a precomputed draw call, the following calls are made:

```
// pass in the parameters required for precomputed draw objects that include
pointer to all vertex streams and the index list. The index for precomputed object
to be created is also passed in.
pPrecompData->initPrecomputedDrawData(…, uint32_t index);
```

Similarly, a vertex and fragment state is created by calling these two functions:

```
// pass in base vertex shader program and the index of the state
pPrecompData->initPrecomputedVertexState(const SceGxmPrgram *vtxBaseProg,
                                         uint32_t index); // for vertex state

// pass in base fragment shader program and the index of the state
pPrecompData->initPrecomputedFragmentState(const SceGxmPrgram *fragBaseProg,
                                           uint32_t index); // for fragment state
```

Corresponding uniform buffers, textures, and the address for the default uniform buffer required by the specific vertex and fragment states are set here, and are later patched if the data is changed.

Scene Updates

After the aquarium scene starts to run, the tutorial sample application needs to make a few updates for each frame. Collision is enabled for the scene, so the collision detection and response are simulated by the simulator, and active objects (fish, play balls instances) accordingly have their associated data updated. This is executed when the following call is made:

```
// need to pass in delta time since previous update
m_simulater.simulate(s_frameDelta/N_SIMULATE_PER_FRAME, 1);
```

The water surface, which is animated based on a sinusoidal wave, is also updated by calling:

```
// need to pass in delta time since previous update
m_waterSurface.animate(s_frameDelta/N_SIMULATE_PER_FRAME);
```

Updating Precomputed Objects

Next the tutorial sample patches the updated uniforms and other data used by the precomputed objects. The updates are done by calling the following for each geometry:

```
// need to pass shader effect pointer and the uniforms to be patched in the default
uniform buffer
m_fishes.updateShaderParams(…);
m_plants[0].updateShaderParams(…);
m_playBalls.updateShaderParams(…);
```

Because parameters to the vertex and fragment programs are updated, only the related vertex and fragment states need to be patched in for each of the instances and variations of the corresponding geometry. The updated default uniform parameters are patched into the memory embedded in the specific precomputed state. For example, for the fish, the tutorial passes in updated skinning parameters and updates the world matrix to the associated shader effect implemented in `shader_util.cpp`. This is done by calling:

```
//get the updated skinning parameter and the world matrix for the instance
SkinningParameter *pSkinningParams = m_pFishData->m_skinningParameter[index];

// pass them as arguments to patching function for the shader effect along with
the address of the default uniform buffer embedded in the vertex state object
shaderEffect->patchPrecomputedVertexShaderParams(pPrecompData->getVertexDefa
ultBufferByInstance(index), pSkinningParams, m_pFishData->m_localToWorld);
```

As discussed later in the Synchronization and Patching section, precomputed objects should not be updated while they are being rendered.

Scene Rendering

The final aquarium scene can be rendered using different configurations of precomputed and nonprecomputed objects. The objects with no associated precomputed objects (aquarium walls and floor, water surface) are rendered using the standard pipeline.

```
// rendered with caustics effect using caustic_map_v.cg and caustics_map_f.cg
m_aquarium.renderFloor(pContext, &g_sceneResources.m_floorCausticEffect,...);

// rendered with transparency using basic_color_v.cg and basic_color_f.cg
m_waterSurface.render(pContext, &g_sceneResources.m_colorEffect, ...);
m_aquarium.renderWalls(pContext, &g_sceneResources.m_colorEffect, ...);
```

Rendering Precomputed Objects

For the geometries with precomputed objects (fish, sea-plants, and playball instances) , the tutorial uses render functions that support using either precomputed or nonprecomputed states, and you can toggle them dynamically. These are rendered as:

```
// rendered using basic_color_v.cg and basic_color_f.cg, with options of using
either precomputed or nonprecomputed states
m_playBalls.render(pContext, &g_sceneResources.m_colorEffect,
    usePrecomptedDataState, // use precomputed draw if true
    usePrecomptedVertexState, // use precomputed vertex states if true
    usePrecomptedFragmentState); // use precomputed fragment states if true

// rendered using skinning_phong_v.cg and skinning_phong_f.cg
m_fishes.render(pContext, &g_sceneResources.m_skinnedShadowEffect,...);

// rendered using static_geom_v.cg and static_geom_f.cg
m_plants[0].render(pContext, &g_sceneResources.m_colladaShadowEffect,...);
```

The configuration of which type of object is rendered for the geometry is defined by the `bool` arguments corresponding to each draw object, vertex state, or fragment state passed in the render function. The implemented render functions accordingly configure the states and issue the specific draw call.

If a draw call uses non-precomputed vertex states, the render function issues commands to reserve the default uniform buffer, copy in uniform data, and set the associated uniform buffer, such as for the skinning matrices for the fish. If a draw call uses precomputed states, all the updated states are already implemented, and you only need to set the precomputed states and issue the precomputed draw call.

To effectively use either of the cases, the tutorial uses the following implementation:

```
// setting up states for precomputed vertex and fragment states, based on the
bool parameters. If using nonprecomputed states, -1 is passed for that state;
otherwise the index of the state to be set is passed
pPrecompData->startPrecomputedRender(context,
    (usePrecompDrawCalls ? idxDrawState : -1),
    (usePrecompVertexState ? idxVertexState : -1),
    (usePrecompFragmentState ? k%NUM_FISH_VARIATIONS : -1));
// If precomputed draw call is to be used, this branch is skipped
if(!usePrecompDrawCalls)
{
    // rendering using standard nonprecomputed draw call that needs setting up
    the vertex streams, indices, and other draw parameters dynamically.
    sceGxmDraw(context, SCE_GXM_PRIMITIVE_TRIANGLES,
SCE_GXM_INDEX_FORMAT_U16,
    vData->m_pGeomIndices, vData->m_indexCount);
}
// This call issues a precomputed draw call if it is enabled. Also, if the fragment
and vertex states are used then these need to be set to NULL to revert to standard
pipeline. For nonprecomputed states these state set ups are skipped.
pPrecompData->endPrecomputedRender(context,
    (usePrecompDrawCalls ? idxDrawState : -1),
    (usePrecompVertexState ? idxVertexState : -1),
    (usePrecompFragmentState ? k%NUM_FISH_VARIATIONS : -1));
```

Note: Because the tutorial sample contains transparent objects (water surface, aquarium boundaries), it facilitates efficient hidden surface removal by rendering all opaque geometries before the alpha blended objects.

Synchronization and Patching

For safe patching of precomputed objects, you need to make sure that the GPU is not rendering the objects when you do the patch. To ensure this, the tutorial uses notification for fragment processing to be written before starting the patching process. The notification mechanism is set by:

```
++m_sceneNotification[g_currentBufferIndex].value;

// stop rendering to the render target
sceGxmEndScene(m_graphicsData.pContext, NULL,
               &m_sceneNotification[g_currentBufferIndex]);
```

If you are using multiple buffers of dynamic precomputed objects, at this point you need to switch to the new buffer, which has its precomputed objects used by the GPU and that are now safe for the CPU to update. This switch is done by:

```
g_currentBufferIndex = (g_currentBufferIndex + 1) % NUM_BUF_COUNT;
```

After the notifications are set and processing for the current frame is started by the GPU, make sure that the CPU waits for the notification from the GPU for completion before patching them with updates for the next frames. This is done by calling the following function before calling `updateShaderParams(…)` for all the precomputed objects:

```
sceGxmNotificationWait(&m_sceneNotification[g_currentBufferIndex]);
```

The patching of the precomputed objects in that buffer will start only after the GPU notification struct for the buffer to be updated is written to.