Tips for optimizing OpenGL ES 2.0 widgets on PNaCl

Published 2014-10-28 | (Compatible with SDK 4.5,5.0,5.1 and 2013,2014 models)

This article describes some tips and possible methods for optimizing OpenGL ES 2.0 applications running on PNaCl platform.

Contents

It is hard to create OpenGL ES game but it is even harder to optimize it. This text introduces a couple of tricks that can make this hard task a little easier. It is obvious that there is no all-in-one way to do it. Concrete optimizations will always depend on particular application and developer. But for sure we can show some tips forming a starting point for this rough path.
This article is designed for developers who have experience with PNaCl and OpenGL ES 2.0 and want to benefit the most from them. This is neither Native Client, nor OpenGL ES 2.0 tutorial so it is assumed that reader knows at least basics of the mentioned technologies. If you are not familiar with them, please refer to the Related Documents. It is strongly recommended to read **https://developers.google.com/native-client/dev/devguide/coding/3D-graphics**, Native Client 3D Graphics Developer's Guide. Take especially a particular look at the last section named Tips and Best Practices.

Developer's Guide

Here you will find tips for optimizing your OpenGL ES 2.0 application on the Native Client platform.

Usage of glBufferData and glBufferSubData
First of all, examine why you want to update VBO manually. Many common operations can be performed using vertex shaders. These are some examples: translation, rotation, scaling of meshes, skeletal animation, morphing, displacement mapping. If you do not have serious reason to update VBO, just do not do it, use vertex shader instead.
If you really have to update buffers' data, do it only if they were changed. Never treat calling glBufferData and glBufferSubData like a harmless habit. Sending updated data through the command buffer always takes time. Try to minimize amount of calls of these functions.

Do not mix glDrawArrays with glBufferData or glBufferSubData
It is tempting to update the buffer just before it is used in drawing out of convenience. If one succumbs to this temptation, one creates the following OpenGL calls sequence:
glBufferSubData
glDrawArrays
glBufferSubData
glDrawArrays
glBufferSubData
glDrawArrays
But why is it a temptation? Update of the buffer can take much time. If draw call is issued, GPU is stalled waiting for buffer update. It simply beats the application performance. It is really slow - rendering can take up to 100 times more than under normal circumstances. To avoid this, use the following call sequence:

glBufferSubData
glBufferSubData
glBufferSubData
//...as much other code as possible...
glDrawArrays
glDrawArrays
glDrawArrays

Textures

glBindTexture binds texture to the current target and texture slot. Texture remains bound until you unbind it or change it for this particular target and slot. So there is no need to bind the same texture again and again before each draw call if the texture remains the same.

glTexParameteri is used to set parameters of textures like filtering, wrapping etc. You set them up just right after texture is loaded. But do those parameters need to be reset every time the texture is bound? The answer is no, don't repeat parameters setting after subsequent glBindTexture calls for this texture.

State changes issued by glBindTexture are always tied with overhead. To minimize their amount you can use a technique called texture atlas. This is a combination of multiple textures put together in one texture, so it can be used to render multiple objects. Texture resource does not change, so you save time on a few glBindTexture calls.

Use texture compression. In the Native Client on Smart TV there is only one texture compression format available: ETC1. Unfortunately it has several limitations: compression ratio is 1:4 and it stores only RGB channels (so it is hard to use it with blended materials). It is better than nothing though, as it improves memory allocation and bandwidth, which are limited resources in Smart TV.

Buffers

You can also save performance by rendering the same buffer a couple of times. Given the fact that it is an expensive operation, glBindBuffer does not need to be recalled after draw call. If you have game objects using the same buffer to render, group them together to minimize glBindBuffer calls. Here are some examples: rectangle representing 2D objects, similar AI agents, ambient objects.

Use GL_STATIC_DRAW with glBufferData (if you don't intend to change the buffer frequently) where possible. This tells GPU to optimize allocation of this buffer, because you promise not to modify it often, or at all. Remember that modification of a static buffer costs more than a dynamic one. Moreover, do not mix the static data with the dynamic data.

Shaders

Change shading program as seldom as possible (by glUseProgram function). You can group renderable objects together by shading programs they use.

If you don't change the shading program you don't also need to call glEnableVertexAttribArray function again. If both rendered buffer and program are not changed, glVertexAttribPointer function does not need to be recalled as well.

State changes

Try to organize your rendering algorithm to minimize state changes issued by glEnable and glDisable functions. As mentioned before, state changes are always tied with overhead.

You may implement OpenGL state tracing mechanism which saves current state of overall state and shaders, textures and buffers states and prevents to unnecessarily call OpenGL functions. It is advisable for performance of your application to find and eliminate duplicating function calls either by special mechanism or manual profiling.

Stalling calls

glReadPixels reads a block of pixels from the framebuffer. By all means try not to use this function. It results in waiting while the GPU is finishing rendering. It also has a significant command buffer overhead. Finally you get powerful performance blocker.

glCopyTexImage2D copies framebuffer's pixels into a 2D texture image. Use FBOs instead of this call.

glTexSubImage2D specifies a 2D texture subimage. Normally, this is not a stalling call, unless you call it on FBO, so use it carefully.

glFlush and glFinish

In the Native Client platform OpenGL commands are not executed in the same process as your application is working on.

Actual OpenGL calls are issued in the special process called GPU Process, because of security reasons (GPU Process checks their validity first). OpenGL commands are batched up in a buffer and sent to GPU Process through RPC mechanism. This command buffer needs to be flushed from time to time. In most cases flush is issued by the Native Client in some OpenGL ES function calls implicitly. But you can also issue flush explicitly. glFlush issues asynchronous flush of the command buffer toward GPU Process. glFinish issues synchronous flush of the command buffer and waits for all buffered OpenGL commands to be executed in GPU Process.
The problem with automatic flushes is their unpredictability. Flush can occur during hard processing in a random OpenGL call, so you may observe temporary performance drop. To prevent this drops force command buffer flush with glFlush or glFinish before hard processing. In order to choose how often issue a flush in your code, you will probably need some testing and profiling.

Blending

Because of internal structure and implementation alpha blending in Smart TV's Native Client platform is particularly expensive. Therefore, you need to pay special attention to alpha blending optimization. First of all, keep alpha blending switched off when it is not currently used. Secondly, avoid rendering big alpha blended rectangles, especially fullscreen rectangles. If you need to create HUD in your game, try to divide HUD geometry to draw as small rectangles as possible - just to avoid big empty spaces (i.e. rendered with 0 alpha value). The third and the most interesting technique is to sort rendered objects: firstly all objects without alpha blending, secondly objects rendered by fragment shader with discard inside and lastly alpha blended objects. This enables early Z clipping - expensive objects with alpha blending will not be rendered at all if they are not visible, but they will be clipped by Z buffer.

Sort

You can benefit from aforementioned tips by implementing an algorithm that sorts calls to OpenGL ES 2.0 to minimize their amount. If your game or application is large and complicated enough, it is recommended to create sorting algorithm that groups usage of shaders, textures and buffers together. The concrete sorting algorithm depends on your needs. Maybe your application's problem is a frequent change of shaders which you have only few. Maybe your 2D game has only one buffer and one shader to render rectangles and the only thing you need to do is group same textures together. Below there is presented some pseudocode of a possible algorithm:

```
void Render()
{
  for (int i = 0; i < renderables.size(); i++)
    renderables[i]->CalculatePriority(); //ex. programIndex << 32 | textureIndex << 16 | bufferIndex, but it depends on yo
ur needs
  renderables.sort();

  for (int i = 0; i < renderables.size(); i++)
  {
    if (currentProgram != renderables[i]->GetProgram())
      ChangeProgram(renderables[i]->GetProgram()); //appropriate calls to OpenGL ES

    if (currentTexture != renderables[i]->GetTexture())
      ChangeTexture(renderables[i]->GetTexture()); //appropriate calls to OpenGL ES

    if (currentBuffer != renderables[i]->GetBuffer())
      ChangeBuffer(renderables[i]->GetBuffer()); //appropriate calls to OpenGL ES

    Render(renderables[i]);
  }
}
```

Batch, batch, batch!

Batch static renderable objects when possible. Batching means grouping different logical objects into one VBO. Remember, that those objects really need to be static. Updating batched dynamic objects requires updating whole or part of VBO, which

becomes unprofitable, because of sending potentially large data through the command buffer. Below there is a batching example: you have a couple of quads that differ only in transformation matrix. It is faster to create one VBO with all objects and issue one draw call, than to have VBO with one quad only, send transformation to shader for each quad and issue as many draw calls. First of all, you benefit from issuing only one draw call. Moreover, in the first case you send vertex buffer only once, while in the second each frame you send matrices multiple times depending on objects' amount. Thanks to this technique you minimize drivers' and command buffer's overhead of each not issued call.

Other

Reduce resolution of a widget. If you really do not have to use 1080p resolution, use, preferably, 720p resolution.
Do not clear color buffer if you do not need to (e.g. you have a skybox).
Use shader precision qualifier wisely: when in doubt, use highp; use lowp for colors; use highp for most vertex operations; use mediump for normals and vectors used in lighting calculations.
Cull invisible geometry by using glCullFace.

Summary

To sum up, the above tips are written to help you optimize your Native Client OpenGL ES 2.0 application. Resources (CPU, GPU, memory) on the TV board are limited, therefore try to minimize OpenGL calls as much as possible. This will save these precious resources for other important calculations your application needs to perform and gain better performance. Feel free to refer to the below links to find more information on PNaCl and OpenGL ES 2.0.

Related Documents

Native Client related

Getting started with NaCl
How to create sample PNaCl application
https://developers.google.com/native-client/overview - Native Client Technical Overview

OpenGL ES related

http://www.khronos.org/opengles/sdk/docs/man/ - OpenGL ES 2.0 Reference Pages
https://developers.google.com/native-client/dev/devguide/coding/3D-graphics - Native Client 3D Graphics Developer's Guide