Optimizing 3D Applications for Platforms Based on Intel® Atom™ Processor

March 2010
Abstract

This paper presents 3D application optimization techniques that are particularly suitable for low-power platforms based on the Intel® Atom™ Processor and the POWERVR* SGX graphics core, such as those featuring the Intel® Atom™ Processor Z5xx Series and Intel® System Controller Hub US15W, and similar future platforms. More specifically, the paper discusses the distribution of tasks between the CPU and the GPU, and the optimization of the interactions between these components. An OpenGL* sample application is provided to illustrate the techniques.
Contents

1 Introduction ............................................................................................................... 4
2 Scene Contents ........................................................................................................... 6
3 Sample Application ..................................................................................................... 7
   3.1 Code Organization .............................................................................................. 9
   3.2 Code Overview ................................................................................................. 10
       3.2.1 Viewpoint Changes ................................................................................. 11
4 Optimization Steps ................................................................................................... 13
   4.1 Initial Version ................................................................................................... 13
   4.2 Pre-Computed World ......................................................................................... 15
   4.3 Vertex Buffer Objects ........................................................................................ 15
   4.4 Texture Atlasing ............................................................................................... 21
5 Further Considerations ............................................................................................. 25
   5.1 OpenGL* ES 2.0 Programmable Pipeline .............................................................. 25
   5.2 Additional Recommendations ............................................................................. 25
6 Conclusion ................................................................................................................ 26
7 Reference List .......................................................................................................... 27

Figures

   Figure 1. Screenshot of the Sample Application ....................................................... 7
   Figure 2. Scene Data Structures ............................................................................... 11
   Figure 3. Setting up the Viewpoint ........................................................................... 12
   Figure 4. OpenGL* Immediate Mode Functions ....................................................... 13
   Figure 5. Using OpenGL* Vertex Buffer Objects .................................................... 16
   Figure 6. Vertex Indices ......................................................................................... 18
   Figure 7. Vertex Buffer Objects Setup ..................................................................... 19
   Figure 8. Texture Atlas Implementation .................................................................. 22
   Figure 9. Infrastructure Classes for Texture Atlasing ............................................. 22

Tables

   Table 1. Source File Versions vs. Optimization Steps .............................................. 10
   Table 2. Performance Improvements ...................................................................... 20
1 Introduction

Platforms based on the Intel® Atom™ processor and the POWERVR* SGX graphics core, such as those intended for the In Vehicle Infotainment market, can deliver a tremendous 3D graphics experience in a very low power envelope. This, however, requires that the application be written so that the platform can operate in its most efficient manner. This involves optimizing communication between the CPU and the graphics core, as well as appropriately distributing tasks between both components.

The graphics core supports 3D acceleration features that allow off-loading tasks from the CPU that the GPU can execute faster. However, the graphics core is also constrained by the power envelope and does not pretend to compete with high-end video cards found in the desktop environment. For this reason it may not be appropriate to offload to the GPU tasks not directly related to 3D scene rendering — especially if a fluent 3D experience is expected. Take, for example, scene geometry creation processing. Even though it is possible to defer a large part of this processing to vertex shader code executed on the GPU, if the geometry or significant portions of it remain static over a large number of frames, pre-computing the geometry on the CPU and sending ready-to-render data to the GPU will probably yield a better balanced load on the platform components than letting the GPU redo this same operation on the same data at each frame.

From an API perspective, OpenGL* allows multiple ways of interacting with the GPU: immediate mode, vertex arrays, display list and vertex buffer objects (VBOs). These, however, exhibit very different performance patterns. On desktop systems, all modes may provide adequate performance for many situations, but on low-power platforms, it is often of utmost importance to select the most efficient API mode.

This paper attempts to demonstrate a set of techniques that can be used to optimize the communication between the CPU and the graphics subsystem, and to achieve a good balance of the tasks between both components. This is illustrated by sample code that evolves from a traditional OpenGL* immediate mode code to a version that takes advantage of the efficiency and the parallelism provided by vertex buffer objects. Although the sample code uses the OpenGL API, the guidelines given can also be applied to applications using the Direct3D* API.

The techniques covered are not specific to embedded or low-power platforms. This paper shows, however, that they are critical in achieving optimal performance on low-power platforms based on Intel Atom processors and POWERVR SGX graphics cores.

Here is a brief summary of the techniques described (or referenced) in this paper and applied in the sample application:

- Use of scene data organization (scene graph) that allows effective reduction of the scene content
- Use of Vertex Buffer Objects (VBOs)
- Optimum usage of VBOs by fully pre-computing all static scene elements
Introduction

- Avoidance of unnecessary state changes
- Front-to-back sorting is not beneficial; favor render state sorting
- Further reduction of state changes via texture atlases
- Use of textures of appropriate size and precision for target system
- Use of mipmaps with GL_LINEAR_MIPMAPNEAREST
- Limit frame rate to what is needed for a good user experience

These techniques helped double the performance of the sample application in terms of achieved frame rate (without frame rate limitation) while dividing the CPU usage by four.
A first consideration is which items are drawn by the application each time the 3D scene is rendered. Even though the GPU can perform clipping, culling, etc., to only display the parts of the scene that are visible through the viewport, each item that the application includes in the scene requires data exchanges and some processing even if it ends up being outside of the visible area, or so small that the user would not be able to distinguish it.

This is not really specific to embedded platforms and should be done for any 3D application, but with the 3D rendering power of modern desktop systems, it is easy to neglect this aspect when designing an application.

Consequently, an initial and very important optimization involves only instructing the GPU to render the elements that are needed for the current scene. As an extreme example, a 3D navigation application should not request the GPU to draw an entire city when the current view only shows a few streets away from the current position.

This optimization is not illustrated in this white paper as it is very application-specific. A common practice is to divide the 3D world into sectors and to only consider for rendering the elements from the sectors that are appropriate for the current position and viewing parameters. Such considerations should be taken into account when designing the scene graph or the scene spatial representation of the application.

Also, the screen resolution of the target system should be taken into account when selecting the size of the textures to be applied to the 3D surfaces. For example, there is no point in using a 1024x1024 texture for items that, under normal condition, will not be wider than a hundred pixels on the screen. Reducing the size of the texture will both improve performance of the graphics pipeline and will also allow your application to fit more textures into the video memory.
3 Sample Application

To illustrate the optimization techniques described in the following sections, a sample application is used. It can be downloaded from [http://edc.intel.com/Go/3405/](http://edc.intel.com/Go/3405/).

The sample application is an extremely simplified version of the animation that a 3D navigation system could generate. It uses a very simple representation of buildings and moves only within a few streets in a very simple manner. Nevertheless it helps demonstrate most of the various optimization techniques that also apply to the more complex and complete applications. Figure 1 shows a screenshot of the application running.

![Figure 1. Screenshot of the Sample Application](image)

The scene rendered by the sample application is a static one and corresponds to the complete 3D world implemented. This does not correspond to what most applications would need, especially taking into account the first optimization recommended in the previous section which requires adding elements to the scene as the viewer moves within the 3D model. However, one needs to understand that these modifications of
the content of the scene are likely to be done at a fairly low rate for most 3D applications. A 3D navigation application can split its map into sectors of appropriate size and use the data of a set of sectors only for some time until the position moves closer to the border of the covered area. At this time data from new sectors needs to be added to the scene composition and data from sectors at the opposite side probably can be removed. As a result, the changes of scene composition happen at a much lower rate than the re-rendering of the scene due to viewpoint modifications (moving as the vehicle moves, turns, etc.).

The application uses a few bitmaps, loaded from files, as textures for the buildings, the pavement, and the roads.

When executed, the application shows a view from within the small city and starts driving along the streets, following a simple pre-defined path (later referred to as the “auto-pilot” mode). The user may also take manual control by simply pressing one of the arrow keys and move freely within the city very much like in First Person Shooter games. There is no collision detection so you can cross any wall.

The following keys control the movements and the view:

- Up and Down arrow keys - Move forward or backwards (also stops the auto-pilot mode if active).
- Left and Right arrow keys - Turn (while moving or not) (also stops the auto-pilot mode if active).
- Page Up, Page Down - Move head upwards or downwards.
- ‘A’ - Resume or restart the auto-pilot mode.
- ‘W’ and ‘S’ - Change the elevation of the viewpoint.
- ‘O’ and ‘P’ - Change the pace of travel.

The sample application has no frame rate limit and simply tries to render as many frames per second as possible. This is useful to evaluate the performance impact of the various optimizations, but is clearly not something a real application should do. Instead, a real application should put an upper limit to its frame rate and let the rendering thread sleep or perform other tasks between consecutive frames in order to remain below that limit. This either allows other tasks to be completed faster or allows the CPU power management features to lower the power consumption of the system. The upper frame rate limit should be set so that the animation appears smooth enough to the user. The actual limit is very much application- and system-specific and is highly dependent on the amount and speed of motion in the animation. Additionally human eye perception is a fairly complex process and no simple formula can determine the adequate number of frames per second. Several computer games have a limit at 30 frames per second while others have no limit.

**Note:** Taking into account the power envelope of the system and its intended usage is, of course, essential: is it more important to have a long battery life, or more fluid animations?

The sample application has been tested on Windows XP and Linux, on a platform using an Intel Atom processor and the Intel® System Controller Hub US15W chipset and
3.1 Code Organization

The sample application is written in C++ but limits its usage of the language to simple data encapsulation and easy-to-use containers. The code was purposely kept simple so that it is easy to read and so it allows the showing of relevant code fragments to illustrate the techniques described in this paper — without requiring a lot of context.

The optimizations are focused on the 3D-rendering aspects only and major emphasis is put on optimizing the way the work is split between the CPU and the GPU and the manner in which instructions and data are provided to the GPU. No attempt has been made at optimizing the code itself, as this would have been very specific to the sample and would have had little relevance to any real application.

Note that the classes defined often publicly expose some of their attributes so as to make the code more concise. Real applications will likely use getter and setter member functions, etc., to better hide their internal data organization and implementation.

The application code consists of five source files and four header files:

- `citytour.cpp` – This is where the bulk of the code resides and where optimizations are implemented. This file is OS-independent.
- `os_lnx.cpp` and `os_win.cpp` – These files provide the implementation of the OS-dependent functions. This covers window and OpenGL rendering context creation, handling of window events, translation of keyboard events for the common code, reading bitmap files, etc. Each file is used only when building for its corresponding target OS (`os_win.cpp` is used on Windows, `os_lnx.cpp` on Linux).
- `os_lnx.h` and `os_win.h` – One of these header files gets conditionally included by `citytour.h` and provides the required definitions, allowing the code to remain OS-independent (with the obvious exception of the two OS-dependent source files mentioned above).
- `world.cpp` – This file provides the function that constructs the complete 3D model of the city. It is OS-independent.
- `pilot.cpp` – This OS-independent file and its accompanying header file, `pilot.h`, implement a basic path description that is used to automatically follow a defined tour in the streets of the small city.

The application comes with a project file for the Microsoft Visual Studio* environment on Windows and a Makefile for Linux. Compiling the code requires the platform-specific base OpenGL header files to be available.

- For Windows, this is the case assuming the Microsoft development environment is installed. The GL extensions header file however (`glext.h`) is not present by default. It may be obtained from [http://www.opengl.org](http://www.opengl.org), but a copy is included in the sample code archive for your convenience.
On Linux, you will need to install the required packages to have the base OpenGL header files. On a Moblin system, these packages are mesa-libGL-devel and mesa-libGLU-devel.

Table 1. Source File Versions vs. Optimization Steps

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Feature</th>
<th>citytour.cpp replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial version. Immediate mode API</td>
<td>citytour_0.cpp</td>
</tr>
<tr>
<td>1</td>
<td>Pre-computed geometry</td>
<td>citytour_1.cpp</td>
</tr>
<tr>
<td>2</td>
<td>VBO with interleaved vertex data</td>
<td>citytour_2.cpp</td>
</tr>
<tr>
<td>3</td>
<td>Texture atlas</td>
<td>citytour.cpp</td>
</tr>
</tbody>
</table>

The various versions of the application are provided in the citytour_<v>.cpp files where <v> is 0, 1, 2 as shown in Table 1. By default, it's the final version that is built, based on the citytour.cpp source file. To build other versions, you can either modify the Makefile or project file, or replace citytour.cpp with one of the citytour_<v>.cpp files.

### 3.2 Code Overview

The OS-dependent code implements the program entry point which calls the Run() function in citytour.cpp. The Run() function implements the complete application flows and performs the following sequence of operations:

1. It creates the application window and the associated OpenGL rendering context by invoking the CreateGLWindow() OS-dependent function.

2. It then invokes the InitGL() function to perform the remainder of the OpenGL configuration. Texture loading form bitmap files is implemented in LoadGLTextures() with support from the OS-dependent LoadBMP() function.

3. It then proceeds to creating the data for the sample’s scene. This is implemented in SetupWorld() from wold.cpp. This function uses the CreateBuilding() and CreateRoad() functions to add to the scene data a series of buildings and an underlying road texture. SetupWorld() also sets the path for the auto-pilot as this is tightly linked with the city layout. The SetupWorld() function will remain unchanged throughout the versions of the application. Only CreateBuilding() and CreateRoad() will be modified to accommodate changes in the underlying data organization required to implement the various optimization steps.

4. The Run() function then enters the main application loop that continuously renders the 3D scene, checks for keyboard input, and modifies the current position and heading according to the keyboard input or the automatic pilot instructions.

Within the main loop, handling of events from the windowing system is delegated to the OS-specific WinSysCheckMessage() function. The function returns True to indicate the user requested to close the window, which causes the main loop to exit. Upon exit
from the main loop, the `Run()` function simply closes the window and releases everything related to the rendering context before exiting.

Modification of the current position and heading based on the auto-pilot path is handled by the `pilot_move()` function, which makes use of information provided by the `Pilot` object to adjust position and heading.

The 3D scene rendering is done in `DrawGLScene()`. The function first sets up the viewpoint, as discussed in the next section, and then proceeds to render the sector.

The `DrawGLScene()` and the `CreateBuilding()` and `CreateRoad()` functions are the main functions that will be modified in order to implement the optimization steps covered in this paper.

As depicted on Figure 2, the scene data consists of a `Sector` class which is meant to represent a sector of the 3D model. The `Sector` is essentially a collection of `Objects` and the `Objects` represent the buildings. At the next level, an `Object` consists in a set of `Surfaces`. A `Surface` consists in a set of triangles with a reference to their associated texture. As was explained earlier, this sample application uses a single sector which represents its entire world.

**Figure 2. Scene Data Structures**

The data for the vertices composing the triangles is held in `vertexData` structures, which simply contain five float values: the x, y and z coordinates of the vertex and the associated u, v texture coordinates.

### 3.2.1 Viewpoint Changes

The natural approach for implementing viewpoint changes when using OpenGL is to use the ModelView matrix. This represents good use of the GPU resources as the necessary transformations would otherwise have to be applied, using the CPU, to all geometry elements at every single change of the viewpoint (location, angle). The GPU is designed specifically for this kind of task and can therefore efficiently off-load the
CPU in this area. Figure 3 illustrates the use of the ModelView matrix in the sample application.

**Figure 3. Setting up the Viewpoint**

```c
glLoadIdentity();          // Reset view

// Head up/down position (PdDn/PgUp)
glRotatef(lookupdown, 1.0f, 0, 0);

// Our heading is such that 0 degrees corresponds to the x-axis.
// OpenGL's default orientation is with z-axis coming out of
// the screen. In order to look into the direction we're heading
// into, we need an extra rotation of 90 degrees around y-axis.
glRotatef(90 + heading, 0, 1.0f, 0);

// Our position within the world
.glTranslatef(-xpos, -altitude, -zpos);
```
4 Optimization Steps

4.1 Initial Version

The initial version of the code (version 0) mostly uses OpenGL immediate mode functions \( \text{glVertex3f()} \) etc., used between \( \text{glBegin()} \) and \( \text{glEnd()} \) pairs, as illustrated in Figure 4.

Figure 4. OpenGL* Immediate Mode Functions

```cpp
// Iterate through all objects, all surfaces of each object
for (std::vector<Object*>::iterator obj_it =
    sector1.objects.begin(); obj_it != sector1.objects.end();
    obj_it++) {
    Object* obj = *obj_it;

    // Position object using temporary ModelView matrix changes
    glPushMatrix();
    glTranslatef(obj->pos_x, 0.0f, obj->pos_z);
    glScalef(obj->scale_x, obj->scale_h, obj->scale_z);
    for (std::vector<Surface*>::iterator s_it =
        obj->surfaces.begin(); s_it != obj->surfaces.end();
        s_it++) {
        Surface* s = *s_it;

        // Use appropriate texture for the surface
        glBindTexture(GL_TEXTURE_2D, texture[s->texture_idx * 3]);

        // Iterate over the triangles making up the surface
        vertexData* v = s->vertex;
        for (int tri = 0; tri < (s->num_verts/3); tri++) {
            glBegin(GL_TRIANGLES);
            glNormal3f( 0.0f, 0.0f, 1.0f);
            glTexCoord2f((*v)[3], (*v)[4]); // u0, v0
            glVertex3f((*v)[0], (*v)[1], (*v)[2]); // x0, y0, z0
            v++;

            glTexCoord2f((*v)[3], (*v)[4]); // u1, v1
            glVertex3f((*v)[0], (*v)[1], (*v)[2]); // x1, y1, z1
            v++;

            glTexCoord2f((*v)[3], (*v)[4]); // u2, v2
            glVertex3f((*v)[0], (*v)[1], (*v)[2]); // x2, y2, z2
            v++;
        }
    }
}
```
These immediate mode functions require an interaction with at least the driver, and possibly the GPU, for almost every single operation. And since many such operations are required to build a 3D scene, rendering the complete scene requires a lot of interactions between the CPU and the GPU, each time only transferring to the graphics subsystem a small subset of the complete scene data. This is highly inefficient and, to make it even worse, the data transferred contains a lot of redundancy in terms of vertex coordinates since adjacent triangles often include common vertices.

The sample application uses triangles to build its 3D model. OpenGL also supports quads but the “native” geometry basic element of the graphics core is the triangle.

If your application is already using a more efficient approach than the immediate mode operation, you may still want to read on because, even when using vertex arrays, or display lists, chances are that the application still sends the data to the graphics subsystem in many more chunks than is optimal. Also, vertex arrays still waste a lot of CPU cycles by repeatedly sending the same data set to the GPU and moving to Vertex Buffer Objects should provide a significant improvement.

Another important thing to notice about the initial version of the code is that the buildings are scaled and positioned according to their attributes by means of an additional pair of `glScale()` and `glTranslate()` API calls. The ModelView matrix is saved before applying these transformations and is restored before moving on to the next object.

This change in the ModelView matrix represents a state change in the GL state machine for every building and a state change often implies a possibly far-reaching flush of the GL pipeline with potentially big impact on performance. This is particularly true for the graphics cores that implement Tile-Based Deferred Rendering (TBDR) as is the case on platforms such as those featuring the Intel® Atom™ Processor Z5xx Series and Intel® System Controller Hub US15W, and will be the case on similar future platforms.

**Note:** Reducing the number of state changes is one of the most important ways to improve the performance of a 3D application.

Other causes of state changes during the rendering of a frame include texture selection (only one texture may be bound at any time), enabling or disabling features (blending, alpha testing, face culling...), setting attributes (e.g., type of fog), defining new light sources or changing material.

When rendering complex scenes, it is particularly important to architect the rendering code so that geometry is sorted based on rendering state. One way to achieve this is to maintain a separate scene graph, sometimes referred to as a “render graph”, that reflects the render state of the scene elements instead of their logical relationships or spatial repartition. Worth noting is that TBDR will not benefit from front-to-back
sorting of scene elements as is typical on desktop systems. It is therefore important to adapt the geometry sorting scheme to the platform.

These issues, and others, are addressed in the following sections.

4.2 Pre-Computed World

To avoid the GL state changes due to ModelView matrix changes at each object during the scene rendering, the final position of all static elements of the scene can be pre-computed. Since the sample scene is completely static and only the viewpoint changes over time, all vertices can be fully pre-computed and the only changes to the ModelView matrix are then due to viewpoint changes that happen between the frames.

This is implemented in the second version (version 1) of the sample application. Here are the major changes:

- The Surface class now has two new methods to position and scale the surface. They simply apply the given offset or scaling factors to the coordinates of the vertices composing the surface.
- The position and scaling attributes have been removed from the Object class as those are applied directly to the surfaces at the time the object is populated. As a result, the Object class has become merely a container for Surfaces and could in fact be removed, which will be done in version 2 of the code.
- In the CreateBuilding() and CreateRoad() functions, the surfaces are translated and scaled once and for all according to the position and dimensions of the element being created.
- The per-object manipulation of the ModelView matrix in the DrawGLScene() function, with the corresponding glPushMatrix() and glPopMatrix() calls, has been removed.

Thanks to this modification, the number of GL state changes needed for the rendering of the scene have been significantly reduced. However, immediate mode functions are still being used and the entirety of the scene geometry data is still transferred to the GPU each time the scene is rendered. The next round of optimization will address these topics.

4.3 Vertex Buffer Objects

As seen previously, the immediate mode GL API requires several API function calls for each vertex and results in transferring the scene data in many tiny chunks. The OpenGL API allows you to assemble data for several vertices into what is called a Vertex Array, and to then refer to this data within one or multiple API calls. This drastically reduces the number of OpenGL API function invocations and corresponding data transfers. Additionally, Vertex Arrays, through the use of a combination of vertex arrays and associated index arrays, provide the means to avoid redundant vertex data in the data transferred to OpenGL. This not only reduces the size of the data set but...
also allows the render pipeline to avoid repeating the same processing for duplicate vertices.

However, when using vertex arrays, the vertex data still resides in the host memory and needs to be transferred to the graphics subsystem repeatedly at each scene redraw, even if the data does not change at each frame. A further and significantly better enhancement is provided by Vertex Buffer Objects (VBO). VBOs are buffers that can be allocated in video memory, loaded with the same geometry data as would be used with vertex arrays, and be referenced from any subsequent operation, for the next frame or any subsequent one, without the need to repeatedly transfer the data. Also, because the data resides in video memory, rendering performance may be significantly improved.

As discussed previously, in applications such as 3D navigation, the application can be designed so that the scene data remains constant for some time. When using VBOs, this means the geometry data can remain in the video memory and be used for many scene rendering iterations as only the viewpoint changes.

Multiple VBOs can be used by an application and are referred to by means of an identifier, which is typically obtained by allocating one or more VBO identifiers using glGenBuffers(). The VBO then needs to be initialized for a specific purpose, vertex attribute array or index array. This is achieved using the glBindBuffer() API.

Data can be loaded into a VBO by means of the glBufferData() and glBufferSubData() functions. Alternatively, it is also possible to map the buffer into host memory, and write directly to it.

The glDrawXYZ() APIs used with VBOs are the same APIs that are used with vertex arrays but where an offset within the VBO is provided instead of a pointer to array. This is illustrated in the code fragment in Figure 5.

When the scene geometry needs to be updated, it is possible to use the glBufferSubData() function to only update portions of the VBO. As previously mentioned, this will not be illustrated by the sample code as it uses a completely static scene.

**Figure 5. Using OpenGL* Vertex Buffer Objects**

```c
void Sector::Draw()
{
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  int stride = sizeof(vertexData);  // = 5 floats

  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer_id[INDX_BUF]);
  glBindBuffer(GL_ARRAY_BUFFER, buffer_id[VERT_BUF]);

  glVertexPointer(3 /* x,y,z */, GL_FLOAT, stride,
                  (GLvoid*)((char*)NULL) );
  glTexCoordPointer(2 /* u,v */, GL_FLOAT, stride,
                    (GLvoid*)((char*)NULL + 3*sizeof(GLfloat)) ); // Skip 3 initial coords
```
for (int tex=0; tex<NUM_TEXTURES; tex++) {
    glBindTexture(GL_TEXTURE_2D, texture[tex * 3]);

    // Draw all elements with current texture.
    // Use offset into index array
    glDrawElements(GL_TRIANGLES, num_indices[tex],
                   GL_UNSIGNED_SHORT,
                   (void *)(idx_offset[tex]*sizeof(GLushort)));
}
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
}

As can be seen, the drawing itself no longer involves iterating through the vertices. Instead, because the geometry has been sorted based on GL state, which in this case means on the texture, the code only iterates through the various textures and issues a `glDrawElements()` for each, using an appropriate offset in the array of vertices that was previously loaded into the buffer. Depending on the total number of vertices and how you organize them, you could also use different buffers for different states, but this will mean additional calls to `glBindBuffer()` and `glXyzPointer()` APIs.

In the parameters given to the `glDrawElements()` function, there is the number of indices in the index array and an offset in this array for the first vertex index to consider. These values result form the composition of the VBO data based on the scene geometry. Similarly, the parameters given to `glVertexPointer()` and `glTexCoordPointer()` functions reflect the way the vertex data was laid out within the VBO. The `stride` parameter represents the number of bytes to skip in order to reach the data of the next vertex. Similarly, a number of bytes to skip from the start of the buffer is specified (as will be seen later in this section, this allows interleaving multiple vertex attributes).

The task of iterating through the geometry elements still needs to be done, of course. But it can be done just once in case of a completely static geometry (the whole scene or maybe only a part of it), or at least once per many frames in other cases. In the VBO version of the sample application, this is implemented in a new `Sector::GenBuffers()` method, which is invoked once the scene geometry is created, i.e., after the `SetupWorld()` function invocation.

In this version, the source data, as loaded into the `Sector` object by `CreateBuilding()` and `CreateRoad()` now consists of surfaces made up of blocks of four vertices (quads), instead of blocks of three vertices (triangles). This is because the index array feature can be used to reference the same vertex data multiple times. For example, the roof of a building (top face) is represented by the following source vertex data:

```
vertexData building_1_top[] =
{
   // Top = 4-5-6-7 (4-5-6 + 4-6-7)
   // x,  y,  z,  u,  v
   { 0.0, 1.0, 0.0,  0.0,  0.0 }, // 4
   { 0.0, 1.0, 1.0,  0.0,  1.0 }, // 5
   { 1.0, 1.0, 1.0,  1.0,  1.0 }, // 6
   { 1.0, 1.0, 0.0,  1.0,  0.0 }, // 7
```
From the four vertices of a face, two triangles are still created, but by using indices as shown in Figure 6, where \( n \) is the index, within the vertex buffer, of the first vertex of the face. For the top face of the building, this is the index of vertex "4”.

**Figure 6. Vertex Indices**

For this face, vertex data is stored for vertices 4, 5, 6 and 7, and with the indices triplets \( n, n+1, n+2 \) and \( n, n+2, n+3 \), triangles 4-5-6 and 4-6-7 are assembled. The order is chosen so the normals of the faces have a consistent orientation for OpenGL to be able to perform back-face culling.

The code for the `Sector::GenBuffers()` method is shown in Figure 7. It starts by counting the number of vertices per texture, and deriving from it the number of indices per texture, as well as the total number of vertices and indices. While it counts these elements, it also stores the offset within the index buffer at which indices for a specific texture start. `GenBuffers()` then allocates temporary buffers to assemble the data for the vertex buffer and for the index buffer.

It then makes a second pass through the source data, copying the `vertexData` structures into the vertex buffer and writing the corresponding set of indices into the index buffer. Once the temporary buffers have been filled with data, the actual VBOs are created and loaded with the data. Finally, the temporary buffers are freed since the data now resides in the graphics subsystem. For an application that would have to load new data sets into the VBOs as the scene composition changes, this data could be preserved in an application cache so that it would not be necessary to regenerate it in case the same sector becomes part of the scene again.

Because the \( x, y, z, u, v \) data from each `vertexData` is simply copied into the vertex buffer, the resulting buffer contains interleaved vertex coordinates and texture coordinates.

**Note:** Using such an interleaved data structure in Vertex Buffer Objects will often be beneficial for the performance of the rendering process as all data pertaining to a
same vertex will have a much higher chance of being available from the GPU cache
during processing.

This interleaved scheme is also the reason for the byte offset provided to the
gTexCoordPointer() function and the value of the stride parameter seen in Figure 5.
The stride parameter tells the GPU that each vertex datum consists of five floats, and
because it starts with the three vertex coordinates, the first texture coordinates only
appear three floats after the beginning of the buffer.

Another approach would have been to first enter all the vertex coordinates and then
the texture coordinates, or even to use two different VBOs. Either of these would
affect this locality of data and hence possibly degrade the graphics core cache
efficiency.

**Figure 7. Vertex Buffer Objects Setup**

```c
void Sector::GenBuffers()
{
  // For the quads data we use, we make 2 triangles out of them
  // and the indices are 0-1-2, 0-3-2
  GLushort quad_indices[] = { 0, 1, 2, 0, 3, 2 };
  const int INDICES_PER_QUAD =
    sizeof(quad_indices)/sizeof(quad_indices[0]);

  // Count total number of vertices
  total_verts = 0;
  total_indices = 0;
  for (int tex=0; tex<NUM_TEXTURES; tex++) {
    // Remember offset into idx array, corresponding to the
    // set of surfaces sharing the same texture
    idx_offset[tex] = total_indices;
    
    for (std::vector<Surface*>::iterator
      s_it=surfaces[tex].begin();
      s_it != surfaces[tex].end(); s_it++) {
      Surface* s = *s_it;
      num_verts[tex] += s->num_verts;
    }
    // We only have quads in our source data
    num_indices[tex] = (num_verts[tex] / 4) * INDICES_PER_QUAD;
    total_verts += num_verts[tex];
    total_indices += num_indices[tex];
  }

  // Generate data for vertex buffer and index buffer
  GLbyte* verts = new GLbyte[total_verts * sizeof(vertexData)];
  GLbyte* indices = new GLbyte[total_indices *
    sizeof(quad_indices[0])];
  int vert_offset = 0;
  GLbyte* ptr = verts;
  GLbyte* iptr = indices;
  for (int tex=0; tex<NUM_TEXTURES; tex++) {
```
for (std::vector<Surface*>::iterator 
s_it=surfaces[tex].begin(); 
s_it != surfaces[tex].end(); s_it++) {
    Surface* s = *s_it;
    // Copy vertex data into final vertex buffer.
    int sz = s->num_verts * sizeof(vertexData);
    memcpy(ptr, s->vertex, sz);
    ptr += sz;
    // Add indices for each quad
    GLushort* idx = (GLushort*)iptr;
    for (int q=0; q<(s->num_verts/4); q++) {
        for (int i=0; i<INDICES_PER_QUAD; i++) {
            *idx++ = quad_indices[i] + vert_offset;
        }
        vert_offset += 4;
    }
    iptr = (GLbyte*)idx;
}

glGenBuffers(2, buffer_id);

glBindBuffer(GL_ARRAY_BUFFER, buffer_id[VERT_BUF]);
glBufferData(GL_ARRAY_BUFFER, total_verts * sizeof(vertexData), 
            verts, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer_id[INDX_BUF]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 
            total_indices * sizeof(quad_indices[0]),
            indices, GL_STATIC_DRAW);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

delete [] verts;  // Buffer data is now in the video memory
delete [] indices;  // So we can delete it

Table 2 shows the frame rate achieved by the three versions of the application 
covered so far and the corresponding total CPU usage. This is on a platform with a 
single-core, hyper-threaded Intel Atom processor, running the Moblin operating 
system.

Table 2. Performance Improvements

<table>
<thead>
<tr>
<th>Code Version</th>
<th>Feature</th>
<th>Frame Rate</th>
<th>System CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Immediate mode API</td>
<td>54 fps</td>
<td>48%</td>
</tr>
<tr>
<td>1</td>
<td>Pre-computed geometry</td>
<td>60 fps</td>
<td>48%</td>
</tr>
<tr>
<td>2</td>
<td>VBO with interleaved vertex data</td>
<td>105 fps</td>
<td>12%</td>
</tr>
</tbody>
</table>
As can be seen from the CPU usage values, the first two versions of the code were CPU-bound (the application is single-threaded and 50% CPU usage represents one of the hyper-threads being fully utilized). The immediate mode API did not allow the processor to feed enough data at once to keep the GPU busy. Quite logically, implementing the pre-computed geometry did not have much impact since the immediate mode APIs were still used, preventing the leveraging of benefits of pre-computation. The slight increase in frame rate is simply due to the removal of the series of glPushMatrix, glTranslate, glScale and glPopMatrix calls.

The version with VBO is where the benefit really shows: the frame rate increased very significantly when at the same time the CPU usage was divided by 4. This means the CPU is now providing enough work for the GPU to operate independently for some time, thereby allowing the CPU to remain idle part of the time. It should be noted that the application is still trying to pump as many frames as possible. By limiting its frame rate to a lower value, a real application could free even more CPU cycles, making them available for processing other than the 3D rendering or lowering system power consumption.

4.4 Texture Atlasing

If an application uses many different textures, the number of vertices that can be drawn by a single glDrawElements() call (or equivalent) can quickly become too low to still exhibit the same benefits from working with VBOs. Such an application would require many calls to glBindTexture() and glDrawElements() to render its complete scene, which can significantly lower performance.

This can usually be addressed, to a large extent, by grouping multiple textures together in what is often referred to as a texture atlas. A texture atlas is a large texture which contains several smaller textures that are often used together in rendering. The texture coordinates can then be adapted to use, for each surface, the appropriate part of the atlas, corresponding to the original single texture. As a consequence, many more surfaces can be rendered by a single glDrawElements() and the corresponding glBindTexture() calls are eliminated.

Several things should be considered when implementing texture atlasing.

- Using the OpenGL API, it is not possible to use a texture from an atlas when the texture needs to be repeated on the surface (GL_TEXTURE_WRAP_S or GL_TEXTURE_WRAP_T set to GL_REPEAT and corresponding texture coordinate greater than 1.0). Trying to do so would result in using a larger portion of the atlas than intended as there is no way to tell the API where the intended texture ends within the atlas. Such textures need to remain regular textures and require their own pair of glBindTexture() and glDrawElements(). One should, of course, still group all surfaces that use the same repeated texture under a single glDrawElements(). Note that it should be possible to overcome this limitation by means of specific code in a fragment shader, but this is outside the scope of this paper.

- When combining texture atlasing with automatically generated mipmaps, the texture for which mipmaps are produced is the complete atlas, and the filtering used when creating the lower-resolution versions of the atlas can cause some of the textures to become polluted with texels from their neighboring textures. There are several ways to address this. The better
approaches don’t rely on run-time mipmap generation but instead use appropriate pre-processing of the textures. For example, creating the various mipmap levels of each texture separately and then assembling them into corresponding atlases avoids this pollution. In the sample application, these problems were avoided by simply leaving empty space between the textures that make up the atlas.

Texture atlasing is implemented in version 3 of the sample application. As shown in Figure 8, the same texture index used since the initial version to reference the various textures is relied upon, but instead of directly using it as an index into the array of GL texture ids, a `Texture` class is introduced as an intermediary that can apply the necessary mapping.

**Figure 8. Texture Atlas Implementation**

The texture index is then used as an index into an array of `Texture` objects. Each `Texture` object holds the index of the underlying GL texture, as well as any scaling and offset parameters to apply to texture coordinates of surfaces using the texture. The mapping of the texture coordinates is implemented by a virtual function of the `Texture` class. For regular textures, it does nothing and for atlased textures, which have their own `AtlasTexture` derived class, it applies the necessary scaling and offsetting to the provided texture coordinates. These parameters, scale and offset for u and v texture coordinates, are obtained from a separate `Atlas` object that is provided to the constructor of the `AtlasTexture` class. This allows for various implementations of the atlasing scheme, through classes implementing the `Atlas` interface, while still using the same `AtlasTexture` class. The corresponding class definitions are shown in Figure 9.

**Figure 9. Infrastructure Classes for Texture Atlasing**

```cpp
class Texture
{
    public:
        Texture() {};

        unsigned int gl_tex_idx;     // Index into tex_id array
    `
The SimpleAtlas class then provides an implementation of the Atlas interface which corresponds to a simple grid-based atlas, in which textures of identical sizes are organized in rows and columns with constant spacing. More advanced atlas schemes could use, for example, mapping data generated automatically by a texture atlas composition tool.

As textures, the application now loads the regular textures that are not part of the atlas — because they require repeating — and the atlas that contains the textures for the rooftop and the various walls of the buildings:

```c++
const char* texFile[] = { "data/pavement.bmp", "data/asphalt.bmp",  
                         "data/atlas.bmp", NULL };  
```
These are loaded in the same way as in previous versions by the same `LoadGLTextures()` function. Then an extra step is introduced, implemented by the `PrepareTextures()` function, which creates the `Texture` and `AtlasTexture` objects, populating the array of `Texture` objects and configuring them appropriately. For the regular textures, this simply means setting the value of the underlying GL texture index while for the `AtlasTexture` objects, it means providing them with an appropriate `Atlas` object at construction time (an instance of the `SimpleAtlas` class, which holds the GL texture index of the atlas as well as the mapping from the index of the texture within the atlas to the offset and scaling information).

The surfaces are still grouped based on the GL texture index because that is what determines when a change of texture is needed while rendering the scene:

```cpp
void Sector::AddSurface(Surface* s) {
    surfaces[texture[s->texture_idx]->gl_tex_idx].push_back(s);
}
```

Finally, the `GenBuffers()` method of the `Sector` class becomes responsible for invoking, for all vertices, the texture coordinate mapping method of the relevant texture:

```cpp
Surface* s = *s_it;
Texture* pTex = texture[s->texture_idx];
...  
vertexData* vd = (vertexData*)ptr;
for (int v=0; v<s->num_verts; v++) {
    pTex->map_text_coord(vd[v][3], vd[v][4]);
}
```

In the case of the sample application, the texture atlasing technique reduced the number of `glBindTexture()` + `glDrawElements()` pairs from 6 to 3. Even though this represents a drop of 50%, it does not translate into a significant performance boost because the initial number of textures was already very low. For a real application that has many textures, and assuming they can be grouped efficiently so that much bigger batches of primitives can be processed by a single `glDrawElements()` call, the difference can become much more significant.

The aim here is to illustrate the technique and to show that, from a code perspective, the changes can be fairly well-isolated.
5 Further Considerations

5.1 OpenGL* ES 2.0 Programmable Pipeline

The sample application accompanying this paper is written using the OpenGL 2.0 API. In the embedded world, however, the OpenGL ES* variants of the API should be considered. Porting an application from the OpenGL 2.0 API to the ES 2.0 variant is relatively straightforward since the latter is defined with the former as a basis.

There is, however, one significant difference due to the fact that OpenGL ES 2.0 no longer supports the fixed-function transformation. Consequently, a vertex shader program has to be provided for implementing the viewpoint-related transformations and a fragment shader is also mandatory in the programmable pipeline of ES 2.0, whereas in OpenGL 2.0, shader programs were optional. It is, of course, possible to write fairly simple shader programs that replicate the fixed functions used by an application.

5.2 Additional Recommendations

[1] contains some additional recommendations for getting the best performance of the POWERVR SGX graphics core. Here are some key elements but the reader should refer to the referenced document for complete and more detailed information:

- Use depth culling and always clear depth buffer at the beginning of each frame (this is done in the City Tour sample code).
- Use mipmaps (done in the sample code). On the POWERVR SGX, bilinear filtering comes at almost no performance cost, but you should limit the mipmappng mode to GL_LINEAR_MIPMAP_NEAREST for best performance. Mipmaps increase texture memory footprint by 33% but allows the graphics core to use a lower-resolution mipmap when possible.
- Consider using compressed textures. The decompression can be done in hardware and the smaller size in memory may actually speed up texture data reading.
- Smaller texture formats (bits per pixel) may bring immediate performance boosts.
- The GPU uses a 32-bit floating point depth buffer, and as a result, higher precision in z-buffering can be obtained by using an inverted depth range.
This white paper presents techniques that help optimize data exchanges between the CPU and the graphics subsystem and strike a balance between the tasks executed on the CPU and those executed on the GPU. The techniques are not novel, but if used as described, their combination should allow many 3D applications to realize better 3D rendering performance as well as improved resource utilization on low-power platforms based on the Intel® Atom™ Processor and the POWERVR SGX graphics core.

When applied on the sample application presented, these techniques helped double the achieved frame rate while dividing the CPU usage by a factor of four.
7  Reference List


Authors

David Verbeiren is a Senior Software Engineer with the Intel Architecture Group at Intel Corporation.

Philippe Lecluse is a Senior Software Engineer with the Intel Architecture Group at Intel Corporation.

Terminology

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>VBO</td>
<td>Vertex Buffer Object</td>
</tr>
<tr>
<td>TBDR</td>
<td>Tile-Based Deferred Rendering</td>
</tr>
<tr>
<td>SGX</td>
<td>POWERVR* SGX is a range of graphics core from Imagination Technologies</td>
</tr>
</tbody>
</table>

About Moblin

Moblin is an open source project that provides an optimized Linux platform for the next generation of mobile devices including netbooks, mobile Internet devices (MIDs), in-vehicle infotainment (IVI) systems, and other embedded devices. Moblin is optimized for Intel® Atom™ Processor-based devices. More information on Moblin is available at moblin.org.