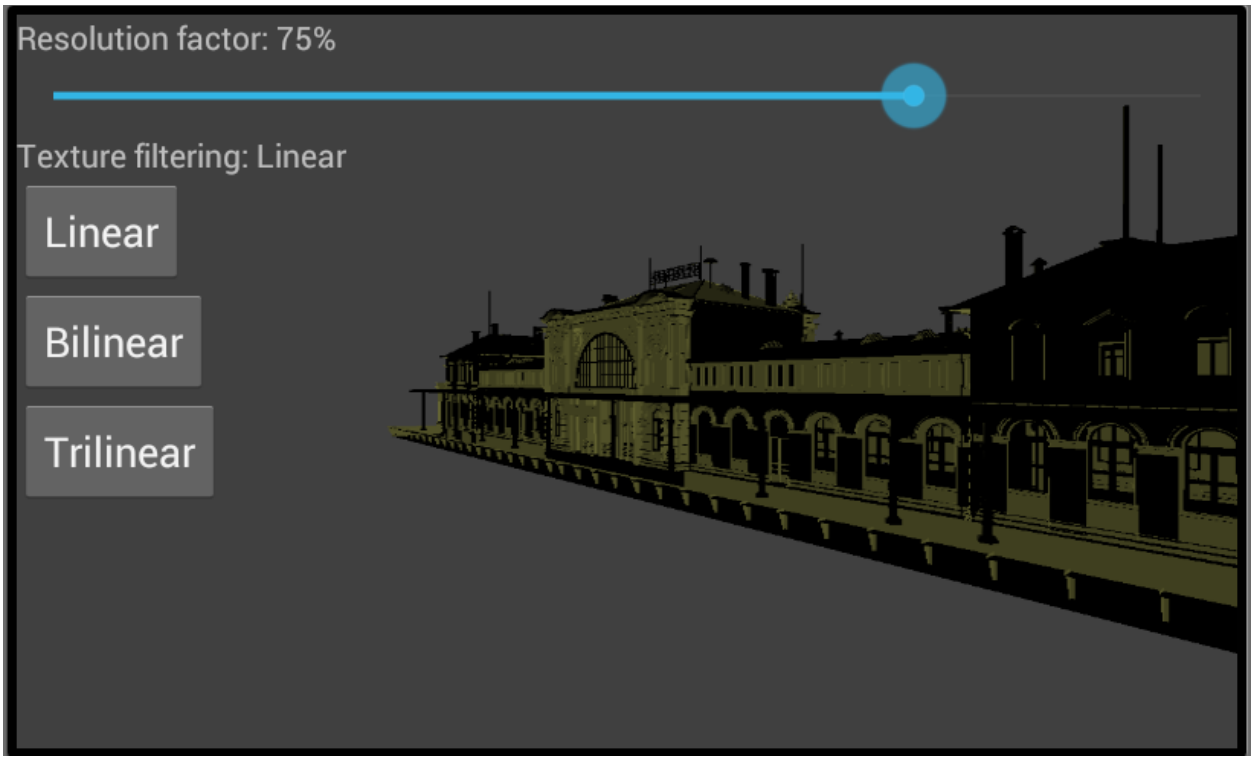
Dynamic Resolution Rendering on OpenGL* ES 2.0



Pixel Processing is Expensive

When doing performance analysis on games and graphics workloads, processing the fragment (or pixel) shader seems to be a major performance bottleneck. Of course, this makes sense because the fragment shader is where lighting calculations, texture sampling, and post-processing effects are computed. Calculating the final color for each pixel on the screen takes a lot of processing power and time and can be prohibitively expensive. This cost is compounded by the fact that mobile platforms are targeting higher resolutions with each new release. Fragment shader invocations will increase when targeting high resolutions. But high resolutions are not the only issue for mobile developers. Targeting devices with different resolutions is another problem. A quick survey of some devices out on the market at the time of this writing shows the variation in resolutions, even for devices running the same OS.

- Apple iPhone* 5: 1136 x 640, PowerVR* SGX543MP3
- Apple iPhone 4S: 960 x 640, PowerVR SGX543MP2
- Nexus* 4, 1280 x 768, Adreno* 320
- Galaxy Nexus, 1280 x 720, PowerVR SGX540
- Motorola RAZR* i, 960 x 540, PowerVR SGX540

- Samsung Galaxy SIII, 1280 x 720, Adreno 225

Clearly targeting not just the ever increasing resolutions but also targeting different resolutions is an issue that game developers are either already dealing with or they will find unavoidable very soon. The other wrinkle is that even as graphics hardware improves it will inevitably be consumed processing more pixels.

Playing the options game

There are true and tried ways to tackle the varying resolutions in games. The easiest way is to draw the scene to the native resolution. Depending on the style of game, you might be stuck with this approach. Or the fragment shader might not be doing enough work to be a performance bottleneck. If you find yourself in this situation, you're mostly set, but you'll still have to make sure your art assets work well across the different resolutions you care about.

A second approach is to decide on a fixed resolution regardless of the native resolution. This will give you the opportunity to tune art assets and shaders to the fixed resolution. However, this might not allow the user to have the best experience with your game.

Another common approach is to allow the user to set the desired resolution at start. This approach creates a back buffer using the player-selected resolution and is useful in combating the varying resolutions problem. It allows the player to select a resolution that works best on their device. You will still have to verify that your art assets work well on the list of resolutions you allow the user to select.

A third approach, the one described in this article, is referred to as dynamic resolution rendering. This is a common technique on console games and higher-end PC games. The implementation described in this article is derived from the DirectX* version described in [Binks 2011] and adapted to work on OpenGL* ES 2.0. With dynamic resolution rendering, the back buffer is the size of the native resolution, but the scene is drawn to an off-screen texture with a fixed resolution. As show in Figure 1, the scene is drawn to a portion of the off-screen texture and that texture is sampled to fill the back buffer. The UI elements are drawn at the native resolution.

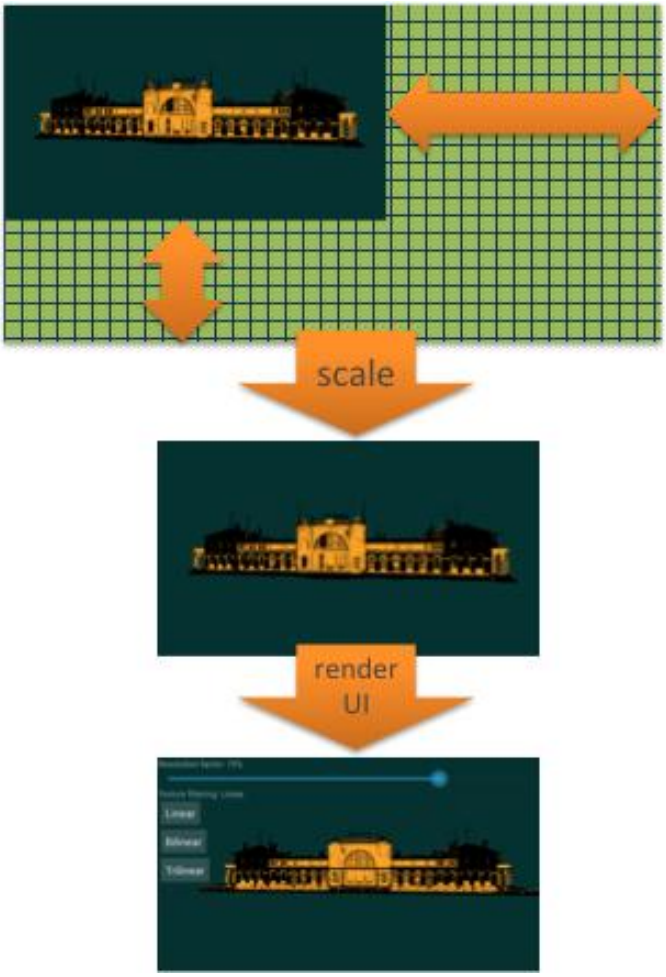


Figure 1. Dynamic resolution rendering

Drawing to an off-screen texture

The first step is to create the off-screen texture. In OpenGL ES 2.0, you create a GL_FRAMEBUFFER with the desired texture size. Here's the code to accomplish this:

```
glGenFramebuffers(1, &(_render_target->frame_buffer));
glGenTextures(1, &(_render_target->texture));
glGenRenderbuffers(1, &(_render_target->depth_buffer));

_render_target->width = width;
_render_target->height = height;

glBindTexture(GL_TEXTURE_2D, _render_target->texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, _render_target->width, _render_target->height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, 0);

glBindRenderbuffer(GL_RENDERBUFFER, _render_target->depth_buffer);
```

```
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16, _render_target->width, _render_target-
>height);

glBindFramebuffer(GL_FRAMEBUFFER, _render_target->frame_buffer);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, _render_target->texture, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, _render_target-
>depth_buffer);
```

The call to glTexImage2D creates the texture we'll be rendering to and the call to glFramebufferTexture2D binds the color texture to the frame buffer. This frame buffer is then bound before rendering the scene as shown here:

```
// 1. SAVE OUT THE DEFAULT FRAME BUFFER
static GLint default_frame_buffer = 0;
glGetIntegerv(GL_FRAMEBUFFER_BINDING, &default_frame_buffer);

// 2. RENDER TO OFFSCREEN RENDER TARGET
glBindFramebuffer(GL_FRAMEBUFFER, render_target->frame_buffer);
glViewport(0, 0, render_target->width * resolution_factor, render_target->height * resolution_factor);
glClearColor(0.25f, 0.25f, 0.25f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/// DRAW THE SCENE ///

// 3. RESTORE DEFAULT FRAME BUFFER
glBindFramebuffer(GL_FRAMEBUFFER, default_frame_buffer);
glBindTexture(GL_TEXTURE_2D, 0);

// 4. RENDER FULLSCREEN QUAD
glViewport(0, 0, screen_width, screen_height);
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Once the scene is rendered, the default frame buffer (back buffer) is bound again. The scene is now saved to the off-screen color texture. The next step is to render a full-screen quad that samples from the off-screen texture. This code shows how to accomplish that:

```
glUseProgram(fs_quad_shader_program);
glEnableVertexAttribArray( fs_quad_position_attrib );
glEnableVertexAttribArray( fs_quad_texture_attrib );

glBindBuffer(GL_ARRAY_BUFFER, fs_quad_model->positions_buffer);
glVertexAttribPointer(fs_quad_position_attrib, 3, GL_FLOAT, GL_FALSE, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, fs_quad_model->texcoords_buffer);

const float2 texcoords_array[] =
{
    { resolution_factor, resolution_factor },
    { 0.0f,              resolution_factor },
    { 0.0f,              0.0f              },
    { resolution_factor, 0.0f              },
};

glBufferData(GL_ARRAY_BUFFER, sizeof(float3) * fs_quad_model->num_vertices, texcoords_array,
GL_STATIC_DRAW);
glVertexAttribPointer(fs_quad_texture_attrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, fs_quad_model->index_buffer );
```

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, render_target->texture);
glUniform1i(fs_quad_texture_location, 0);
glDrawElements(GL_TRIANGLES, fs_quad_model->num_indices, GL_UNSIGNED_INT, 0);
```

The resolution factor

Most of the snippets of code shown above are OpenGL setup code. The interesting sections are the lines that use the variable resolution_factor. This value of resolution_factor determines what percentage of the width and height of the off-screen texture to draw to and then sample from. Setting the section of the off-screen texture draw on is very simple and accomplished by the call to glViewport.

```
// 1. SAVE OUT THE DEFAULT FRAME BUFFER

// 2. RENDER TO OFFSCREEN RENDER TARGET
glBindFramebuffer(GL_FRAMEBUFFER, render_target->frame_buffer);
glViewport(0, 0, render_target->width * resolution_factor, render_target->height * resolution_factor);

/// DRAW THE SCENE ///

// 3. RESTORE DEFAULT FRAME BUFFER

// 4. RENDER FULLSCREEN QUAD
glViewport(0, 0, screen_width, screen_height);
```

After binding the frame buffer, the call to glViewport sets the area to draw based on the width and height. This is then reset to the native resolution to draw the full-screen quad and the user interface. To sample only the section of the off-screen texture that has been updated, set the texture coordinates for the vertices of the full-screen quad. The following code does the trick:

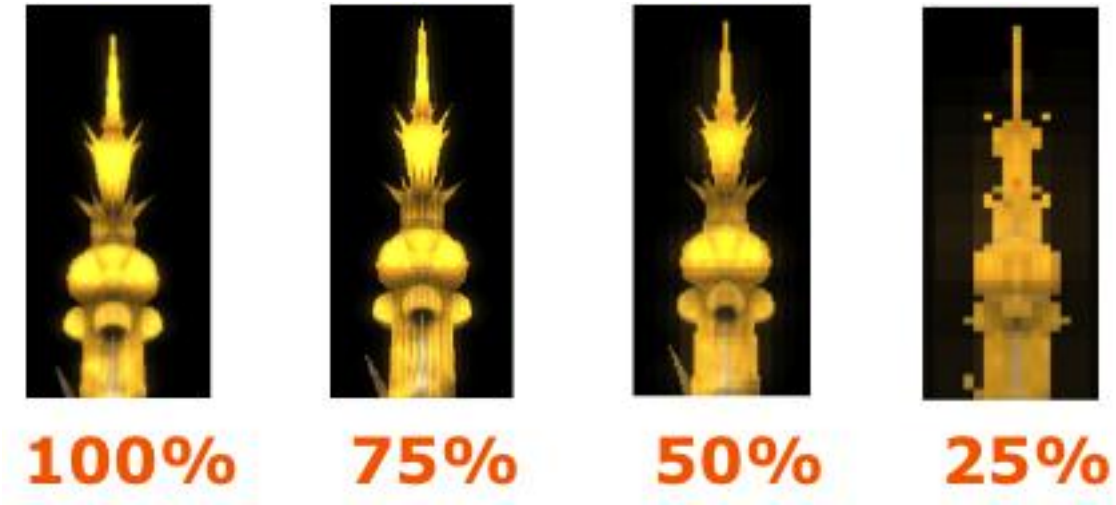
```
glBindBuffer(GL_ARRAY_BUFFER, fs_quad_model->texcoords_buffer);

const float2 texcoords_array[] =
{
    { resolution_factor, resolution_factor },
    { 0.0f,              resolution_factor },
    { 0.0f,              0.0f              },
    { resolution_factor, 0.0f              },
};

glBufferData(GL_ARRAY_BUFFER, sizeof(float3) * fs_quad_model->num_vertices, texcoords_array,
GL_STATIC_DRAW);
```

Benefits of using dynamic resolution

Once the setup is done, the scene will be saved to the off-screen texture and rendered to the screen on a full-screen quad. The actual resolution the scene is rendered to is no longer tied to the native resolution. The amount of pixels processed for the scene can be changed dynamically. Depending on the type and style of game, the resolution can be reduced considerably without much image degradation. Here are a few examples of the sample at various resolution scales:



In this particular case, the resolution can be lowered to somewhere between 75% and 50% before it begins to noticeably degrade the image. The main graphical artifacts that will begin to appear are aliased edges. For this particular case, drawing at 75% of the native resolution is acceptable but, depending on your game, you could go with 25% for an interesting art style.

Dynamic resolution rendering obviously provides a clear way to reduce the number of pixels processed. But, it also provides a way to do more work per pixel. Because you're no longer rendering at full resolution, the fragment shader invocations has been reduced allowing for more work to be done each time it's executed. To try to keep the sample code clear and readable, the fragment shader is very simple. As a developer, keeping a balance between performance and image quality is one of the most challenging tasks.

Our implementation

Figure 2. Implementation details

The implementation available for download contains only the Android* project but is laid out in the format presented in Figure 2 to allow for easy extensibility to other mobile operating systems. The core of the project is written in C, targets OpenGL ES 2.0, and requires the Android NDK. Interestingly enough, C is a very good option for cross-platform development. System abstraction refers to file I/O and other functionality that may be OS dependent.

Conclusion

Dynamic resolution rendering is a good option to solve multiple issues related to screen resolutions on mobile devices. It gives developers and users more control over the performance versus image quality ratio. Tuning this ratio will also involve understanding the overhead of implementing dynamic resolution rendering. Creating a render target and switching render targets every frame will add to frame time. Understanding and accounting for this overhead will help you decide whether this technique is appropriate for your game.

References

[Binks 2011] Binks, Doug. “Dynamic Resolution Rendering Article”. http://software.intel.com/en-us/articles/dynamic-resolution-rendering-article

Notices

DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel and the Intel logo are trademarks of Intel Corporation in the US and/or other countries.

*Other names and brands may be claimed as the property of others.