Showing ads in OpenGL ES context

This article explains how to manually fetch banner advertisements from Inneractive, display them on the surface of a 3D cube using OpenGL, and then open a browser when a cube's side is clicked.

10 Jun 2012

Introduction

There are several ways to make money with your application. One that is currently quite interesting, especially for smaller developers, is advertising. Currently, there are no built-in advertisement platform for Symbian devices, but there is a third-party solution officially recommended by Nokia: Inneractive. They provide a wide variety of different SDKs which can be quite easily be integrated for your application.

While Inneractive allows you to apply normal 2D advertising to your 3D app, there is no support for "3D" advertising - using an advert in a custom way within 3D - content (for example as a texture on your 3d shape like in this example). However, they do provide a simple and robust HTTP-protocol for fetching advertisement data, which can be used for this purpose. This article/example deploys that SDK and explains how to use it in a QtOpenGL Symbian^3 application.

The example application

IAEngine

The Inneractive's server SDK is completely implemented in this class. The main principle is that the object owns one QNetworkAccessManager, which it uses to fetch new advertisements to be stored in-class linked list of `IACapsule`'s which will contain the advertisement information. `IAEngine` is meant to be as simple as possible as seen from it's very simple interface:

```
bool requestAd();
int getCapsuleCount();
IACapsule *getCapsule( int index );
void releaseCapsules();
```

`requestAd()` starts a fetching of a new advertisement IF NO REQUEST currently exists (only one at the time). `getCapsuleCount` returns the size of a in-class `IACapsule` list; how many advertisements we currently have stored. `getCapsule` returns a requested capsule by index for an application to use anyway it wants and `releaseCapsules` destroys the whole list.

When a new advertisement is requested, `IAEngine` sends a HTTP request for it and waits for answer. When the answer arrives, new `IACapsule` is built and filled with the information of the reply. The actual advertisement banner is not contained by the initial reply so another HTTP-requedst must be made to fetch it. This means that the `IACapsule` is alive and available before it's banner image is fetched. When using the advertisements, an application should take this to consideration as well.

More detailed description of Inneractive's server API can be found as a PDF document from their webpages after creating an account.

IACapsule

The `IACapsule` is just a holder for a single advertisement's information. `IAEngine` creates these and fills them when it's requests complete. It's properties are public for easy access.

(C) Copyright Nokia 2014. All rights reserved.

```
bool m_failed;              // true if some requests for this ad have been failed
QString text;               // Text string associated with this Ad
QString targetUrl;          // Url to launch when this Ad is "clicked"
QString imageUrl;           // Url to image (banner) of this Ad
QImage m_image;             // Fetched and decoded image for displaying
```

XMLE

TODO: This will be changed to Qt SDK's own XML parser. For fast developement purposes I've been using my own, simplest possible XML parser **XMLE**. The code is as platform independent as possible (It has Qt bindings only to QByteArray and QIODevice). It parses small XML chunks very fast and in a very practical memory structure. For this example, **XMLE** is used for reading the specific information from the data required from Inneractive's server API.

Widget

"As-simple-as-possible" `QGLWidget` based OpenGL ES 2.0 cube-renderer with minimal vertex / fragment shaders. The most complex part of it is the quad-picking required for knowing which side of the cube user has clicked. It will be explained in separate chapter later.

`Widget` owns a `IAEngine` which it calls to update it's advertisements until the amount of capsules reaches 6 (each side of the cube have it's own advertisement. The advertisements are burned in a single QImage with QPainter so only one texture is used for the entire cube. This `QImage` / GL texture is updated whenever advertisement list is changed. The advertisements are placed in a texture from top to bottom each of them using the whole width of the texture.

Picking a quad

Since this example is working with a cube. I've decided to use quads in the picking instead of triangles. However the princible can be easily extended to triangles. Since this example only has 6 quads, the picking code is not optimized but tried to keep as simple as possible.

The main princible is that when **click** arrives application does similar transformation on CPU side for the cube's vertices than OpenGL ES does when rendering. These screen coordinates are then used to see if the **click**-coordinate is inside a quad or not.

Deploying view / projection transformation manually

The vertex shader in this example uses separate orientation and projection matrices to make the transformation more clean. The vertex - shader looks like this:

```
attribute highp vec4 vertex;
attribute highp vec2 uv;
uniform mediump mat4 matrix;
uniform mediump mat4 proj;
varying highp vec4 frag_pos;
varying mediump vec2 texCoord;
void main(void)
{
 frag_pos = matrix * vertex;
    texCoord = uv;
    gl_Position = proj*frag_pos;
}
```

The corresponding transformation can be done like this:

```
void Widget::transformVector( QVector3D source, QVector3D &target )
{
    QVector4D trans = m_projection * (m_orientation*QVector4D( source, 1.0f));
    target = QVector3D( trans.x() / (NEARPLANE+trans.z()) * width()/2 + width()/2,
                        -trans.y() / (NEARPLANE+trans.z()) * height()/2 + height()/2,
```


(C) Copyright Nokia 2014. All rights reserved.

```
                        trans.z() );
}
```

Backface culling

There are several methods implementing backface culling. I have always used just a 2D cross-product which, in my opinion, is the best and simplest:

```
QVector3D tTemp[4];  // Quad screen coordinates
 // Any 2D vector between 2 points of the quad
float tx1 = tTemp[3].x() - tTemp[0].x();
float ty1 = tTemp[3].y() - tTemp[0].y();
 // Any other 2D vector between 2 points of the quad
float tx2 = tTemp[1].x() - tTemp[0].x();
float ty2 = tTemp[1].y() - tTemp[0].y();
 // Cross product
float crosstest = tx1*ty2 - ty1*tx2;
```

if crosstest is below zero, the quad is facing away from the camera. Other **backface culling** methods can be found around from the Internet.

Inside a quad

The principle used by this example can be extended very easily to cover all convex polygons and quite easily to cover convex-polygons as well. In the name of simplicity, the following function only works with quads.

```
bool Widget::pointInsideQuad( float x, float y, QVector3D *quad )
{
    int edgecount=0;
    float xedges[2];
    int p1, p2;
    int point=0;
    const int edgePoints[] = {0,1,2,3,0,2,1,3};

    while (point<4) {
        p1 = edgePoints[point*2+0];
        p2 = edgePoints[point*2+1];

        // Order the points so p1.y is always smaller than p2.y
        if (quad[p1].y()>quad[p2].y()) {
            int temp = p2;
            p2 = p1;
            p1 = temp;
        }

        // y between the points.
        if (y>=quad[p1].y() && y<quad[p2].y() ) {
            float ylength = (quad[p2].y() - quad[p1].y());
            if (ylength==0.0f) ylength = 0.0000001f;    // Prevent division by zero
   // Interpolate xedge according y
            xedges[edgecount] = quad[p1].x() + (quad[p2].x() - quad[p1].x()) *
                    ( y-quad[p1].y() ) / ylength;
            edgecount++;
            if (edgecount==2) break;
        }
```


(C) Copyright Nokia 2014. All rights reserved.

```
        point++;
    }
 // If the coordinate is between left and right edges, it's inside the quad.
    if (edgecount==2) {     // must be 2
        if (xedges[0]<xedges[1]) {
            if (x>=xedges[0] && x<xedges[1]) return true;
        } else {
            if (x>=xedges[1] && x<xedges[0]) return true;
        }
    }
    return false;
}
```

Function interpolates each edge of a quad according the points y -coordinate. If y is between edge lines a xedge is calculated by interpolating the edge's x-coordinate according the y. After x-edge calculation, if there are two of them, the x-coordinate is checked weither it's between these x edges or not. If it is, coordinate is inside the quad.

Downloads

- Media:adcube.zip - Symbian installation file
- Media:adcube_installer.sis - source files

Summary

It is not difficult to implement a custom advertisement-fetching nor make textures out of the banners. The challenging job is to decide how to use advertisements appropriately - so that they don't impact the applications main functionality. The "custom advertising" - part (IAEngine.h / IAEngine.cpp and IACapsule.h) can be taken from this example and used in any kind of app that needs to display a custom ad.

(C) Copyright Nokia 2014. All rights reserved.