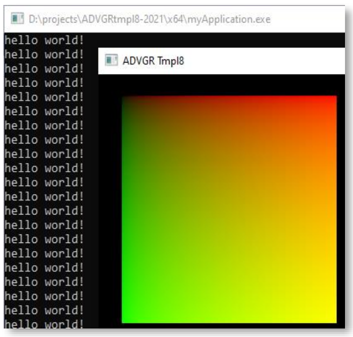
“Getting Started with the ADVGRtmpl8-2021”

This template is only the current final evolution of a long list of ‘templates’, that started with EasyCE, a minimalistic code base for writing Windows CE games / graphics applications without worrying about OS base code. It evolved though Tmpl8 in various versions for IGAD, then UU, then IGAD again, and in the meantime it has been used to start virtually all my personal mini-projects.

To use the template:

- you simply extract it from the zip file to a directory of your choice
- you open the .sln file using Visual Studio.

At the time of writing, Visual Studio 2022 Community Edition is an excellent choice. Get it for free, and install it using the default options, and you’re good to go.



```
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
```

The magic happens in game.cpp:

```
#include "precomp.h"
#include "myapp.h"

TheApp* CreateApp() { return new MyApp(); }

// -----------------------------------------------------------
// Initialize the application
// -----------------------------------------------------------
void MyApp::Init()
{
    // anything that happens only once at application start goes here
}

// -----------------------------------------------------------
// Main application tick function - Executed once per frame
// -----------------------------------------------------------
void MyApp::Tick( float deltaTime )
{
    // clear the screen to black
    screen->Clear( 0 );
    // print something to the console window
    printf( "hello world!\n" );
    // plot some colors
    for( int red = 0; red < 256; red++ ) for( int green = 0; green < 256; green++ )
    {
        int x = red, y = green;
        screen->Plot( x + 200, y + 100, (red << 16) + (green << 8) );
    }
    // plot a white pixel in the bottom right corner
    screen->Plot( SCRWIDTH - 2, SCRHEIGHT - 2, 0xffffff );
}
```

For Advanced Graphics, your job is to turn this into a ray tracer. The default example code shows you the basic low-level ingredients for this:

- A pixel is plotted using screen->Plot(x, y, color).
- The size of the screen can be obtained from SCRWIDTH and SCRHEIGHT.
- A ‘color’ is a 32-bit unsigned value, where red starts at bit 16, green at 8 and blue at 0. Each color component has a range of 0..255.
- You can write debugging info to the text window using printf.

The easy part is now to make for loops that loop over the pixels of the screen. The hard bit is to calculate this color of each pixel, using ray tracing.

A ray tracer ideally operates on floating point color values. The template does not facilitate this out-of-the-box. To use a `float3` pixel buffer, we first create an array for it at global scope:

```
float3 frame[SCRHEIGHT][SCRWIDTH];
```

Each pixel is now addressed as `frame[y][x]`. Note the swapped x and y: this ensures that subsequent x positions are neighbors in memory, which helps performance.

Converting pixels from `float3` to `uint32` can now be done as follows:

```
const float r = frame[y][x].x, g = frame[y][x].y, b = frame[y][x].z;
const uint ir = min((uint)(r * 255), 255u);
const uint ig = min((uint)(g * 255), 255u);
const uint ib = min((uint)(b * 255), 255u);
Plot(x, y, (ir << 16) + (ig << 8) + ib);
```

This code converts each color component from the 0..1 range, which is commonly used in floating point color, to the 0..255 range, which fits the 8-bit-per-component nature of integer color values. It then casts the result to the correct type, to prevent warnings from the compiler. And finally, the (unsigned) integer values are clamped to the range 0..255 (unsigned). The const keyword is a small hint to the compiler: we won't change these variables after creating them.

Useful things

Besides the `float3` type (which you can find in precomp.h), you can use other vector types: `float2`, `float4`, `int2`, `int3`, `int4`, as well as a 4x4 matrix type `mat4`, and a quaternion class `quat`. An axis aligned bounding box class `aabb` is also provided. Please refer to precomp.h for an overview of operators and functions.

In the same file you will find the class `JobManager`, which you can use to run your code on multiple CPU cores. A quick overview of how it is used:

Do once (e.g. in Init), to initialize the job system:

```
JobManager::CreateJobManager( 8 /* your logical core count */ );
```

Then, for the actual parallel code:

```
JobManager* jm = JobManager::GetJobManager();
for( int i = 0; i < jobCount; i++ ) jm->AddJob2( &theJob[i] );
jm->RunJobs();
```

Here, `theJob` is an array of objects of a class derived from `Job`, which must implement `Main`():

```
class theJob : public Job { public: void Main() { /* work */ }; }
```

A high-resolution timer is also provided. See struct `Timer` for details. A timer is created in an arbitrary scope and queried using its `elapsed` method:

```
Timer myTimer;
for (int i = 0; i < 10; i++)
{
    myTimer.reset();
    // ... do something ...
    printf( "iteration took % f milliseconds.\n", myTimer.elapsed() * 1000);
}
```

GPGPU

The template provides OpenCL support to deploy the GPU in your calculations. To use it, first instantiate a `Kernel`:

```
Kernel myKernel( "gpucode.cl", "mainFunction" );
```

The first instantiation will initialize an OpenCL context, load the specified OpenCL source file and compile it. After that, you can pass data to the kernel and run it:

```
myKernel->SetArgument( 1, 3.1415f ); // set the second argument of the kernel function
```

Often, arguments are buffers. Using them works as follows:

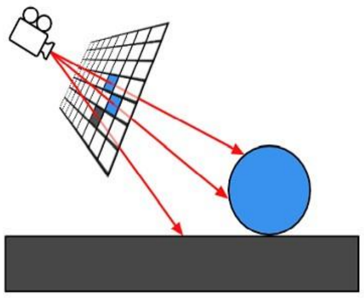
```
static uint hostData[1024];
static Buffer buffer( 1024, Buffer::DEFAULT, &hostData );
buffer->CopyToDevice();
myKernel->SetArgument( 0, buffer ); // set the first argument of the kernel function
myKernel->Run( 1024 /* start 1024 threads */ );
buffer->CopyFromDevice();
```

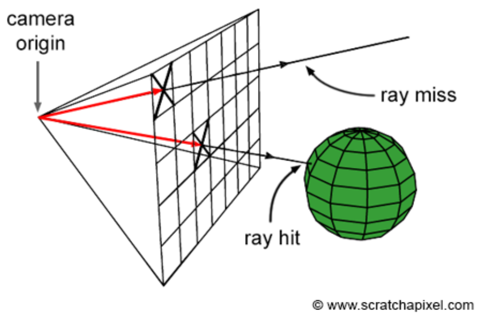
A full OpenCL tutorial is outside the scope of this document. A tutorial that was written for the Concurrency course is still available here. Sadly it is based on a different template (for C#). If you want to see an example of OpenCL used in the `ADVGRtmpl8`, please refer to the voxel template on GitHub.

Go Forth and Code

That should do the job for now; if you have any questions do not hesitate to contact me:

bikker.j@gmail.com / j.bikker@uu.nl




camera
origin
ray miss
ray hit
© www.scratchapixel.com