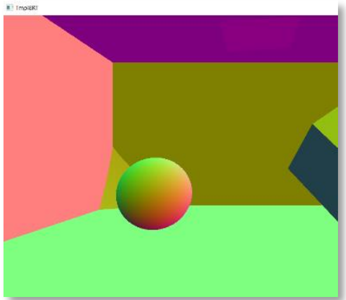
"Getting Started with the tmpl8rt-2022"

This template is a special version of Tmpl8 - he 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 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. In practice, it is great as a basic starting point, but very limited at the same time.

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, install it using the default options, and you're good to go.



From here on:

You're welcome to inspect any file in the project. The interesting content however is in `renderer.cpp`. This file implements a minimal ray tracer, with a stationary camera and a hardcoded scene. In `Renderer::Tick`, the ray tracer loops over lines of pixels:

```
// lines are executed as OpenMP parallel tasks (disabled in DEBUG)
#pragma omp parallel for schedule(dynamic)
for (int y = 0; y < SCRHEIGHT; y++)
{ … }
```

Like the comment says, lines may be executed in parallel, for better performance. Each line is processed by two loops. The first one does the actual ray tracing, via `Renderer::Trace`:

```
// trace a primary ray for each pixel on the line
for (int x = 0; x < SCRWIDTH; x++)
    accumulator[x + y * SCRWIDTH] =
        float4( Trace( camera.GetPrimaryRay( x, y ) ), 0 );
```

The result of this is a floating point color for each pixel. This needs to be converted to something that can be shown on the screen, which happens in the second loop:

```
// translate accumulator contents to rgb32 pixels
for (int dest = y * SCRWIDTH, x = 0; x < SCRWIDTH; x++)
    screen->pixels[dest + x] =
        RGBF32_to_RGB8( &accumulator[x + y * SCRWIDTH] );
```

From here, you're on your own. Explore what happens in `Renderer::Trace` (spoiler: not much), and follow the code path into `scene.h`, which you can find in the `template` folder. The scene is entirely hardcoded, which is great for speed: a high-end PC should get close to 1 billion ray/scene intersections per second. This will come in handy once we get to path tracing.

A few more things about handling pixels in the template:

- 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`.

From here: draw what you need using `screen->Plot` and other `Surface` methods, handle keys and mouse input using the methods of the `Renderer` class (see `renderer.h`) and add .cpp and .h files to extend and structure your project.

Basic math classes can be found in `precomp.h` (starting at line 189). Here you will find `float2`, `float3`, `float4` as well as `int` and `uint` counterparts, with an extensive set of operators. There are also basic classes for storing bounding boxes and for matrix calculations. As with the rest of the template, this serves as a basis; you may find it desirable to add some code of your own depending on what your project needs.

GPGPU

The template provides OpenCL support to deploy the GPU in your calculations. It's use is demonstrated in the #if 1 / #endif block in myapp.cpp:

```
static Kernel* kernel = 0;         // statics should be members of MyApp of course.
static Surface bitmap( 512, 512 ); // having them here allows us to disable the OpenCL
static Buffer* clBuffer = 0;       // demonstration using a single #if 0.
if (!kernel)
{
    // initialize OpenCL; compile and load kernel "render" from file "kernels.cl"
    Kernel::InitCL();
    kernel = new Kernel( "cl/kernels.cl", "render" );
    // create an OpenCL buffer over using bitmap.pixels
    clBuffer = new Buffer( 512 * 512, Buffer::DEFAULT, bitmap.pixels );
}
// pass arguments to the OpenCL kernel
kernel->SetArgument( 0, clBuffer );
// run the kernel; use 512 * 512 threads
kernel->Run( 512 * 512 );
// get the results back from GPU to CPU (and thus: into bitmap.pixels)
clBuffer->CopyFromDevice();
// show the result on screen
bitmap.CopyTo( screen, 500, 200 );
```

The code demonstrates the most important steps in writing GPGPU code: loading and compiling a kernel, creating buffers to pass data between 'host' and 'device', setting kernel arguments, executing a kernel on the device, and retrieving data from device to host.

A full OpenCL tutorial is outside the scope of this document. If you want to see an example of OpenCL used in the ADVGRtmpl8, please refer to the voxel template on GitHub.

Note that unlike plain OpenCL, the template allows you to use #include files in your OpenCL code. The most common use case for this is the common.h file, which gets included by myapp.cpp (via precomp.h) and the example kernels.cl file, to share the default screen resolution between host and device code.

The template also makes some definitions available to your OpenCL code:

- ISNVIDIA will be defined if your code is running on NVIDIA hardware;
- ISAMD and ISINTEL provide the same info, but for AMD and Intel;
- Use ISAMPERE, ISTURING, ISPASCAL to write code specific to an NVIDIA architecture.

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 / bikker.j@buas.nl