

“Getting Started with the tmp18-2022”

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 [Tmp18](#) 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. Good for teaching. 😊

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 (versions 2019 and later).

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.

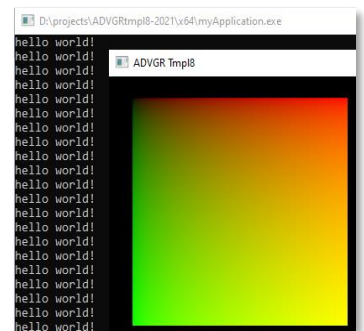
The magic (as seen on the right) happens in game.cpp:

```
#include "precomp.h"
#include "game.h"

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

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

// -----
// Main application tick function - Executed once per frame
// -----
void Game::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, 0xffffffff );
}
```



The default example code shows you the basic functionality implemented by the template:

- A window is opened.
- 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 your own images using `screen->Plot` and other `Surface` methods, handle keys and mouse input using the methods of the `Game` class (see `game.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 191). Here you will find `float2`, `float3`, `float4` as well as (a.o.) `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.

Advanced users may benefit from the integration of `OpenCL`; see the `GPGPU` section later in this document. The math classes are designed to work well with the `OpenCL` functionality.

Useful things

In the `precomp.h` file you will also 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 `Game::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. Here is an example of its use:

```
static Kernel* kernel = 0;           // statics should be members of Game 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)
{
    // prepare for OpenCL work
    Kernel::InitCL();
    // compile and load kernel "render" from file "kernels.cl"
```

```

    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 tmp18, 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 / bikker.j@buas.nl

*: the use of GPGPU is totally optional and only provided for your enjoyment.