CS3211 project 2 - Raytracing: implementation and analysis

February 29, 2016

Due 5pm on Friday April 15, and worth 30% of your course mark.

Part 1: Develop an original raytracing program, which runs both on a GPU and CPU platform, in a browser. You will be doing this using a subset of Javascript, and using GPU.js (GPU accelerated Javascript `http://gpu.rocks/`). In the assessment of your raytracing program, we will give marks for completion (i.e. how much you completed), and for your use of any specific techniques that you use to improve the behaviour of your program (beyond the obvious one-pixel-per-thread in the demo program given to you). We expect your code to be no more than 150-1500 commented lines. The key challenges in ray tracing are load balancing (some pixels take far more work than others) and not swamping whichever processor holds the final picture. In the GPU based technique we will use, this second concern largely disappears, as we work directly on the final displayed canvas(es).

Part 2: Reflect on your program, explaining and evaluating the behaviour, and techniques you used to improve the speed, workload reduction, and/or accuracy of your implementation. Measure and tabulate your results, and explain them with respect to the hardware, and the environment. We expect this write-up to take no more than 10-15 pages.

1 Part 1

We want you to develop a GPU.js based raytracing implementation, with well commented source code. Your program should perform raytracing on a supplied scene description (described below), displaying the result in a web browser, and allowing easy comparison with the CPU version, as in this 2D sample demo:

`http://www.comp.nus.edu.sg/~hugh/CS3211/demo.html`

This sample program has a button to switch between CPU and GPU, and displays an approximate number of frames per second. Your program may look nicer, and have extra options, but this task is not about the look of your web page, it is about the parallelized raytracing component of your program.

You can grab the sources of `demo.html`. The core part is that GPU.js supplies a restricted mechanism to define functions which compute a single float (32 bit) value. This mechanism is supplied options:

1. If you call the mechanism with the option "`mode: cpu`", you get a function which will run on a CPU. With "`mode: gpu`", you get a function which runs multiple threads on the GPU (a *kernel*).
2. Another option defines a target matrix into which results are to be placed. If you call the mechanism with the option "`dimensions: [800,600]`", you get a function which will iterate over all elements of an 800x600 matrix if run on the CPU. If run on the GPU, each thread operates on a single element (of the whole matrix), with co-ordinates (`this.thread.x`,`this.thread.y`).
3. A third option specifies if you are going to be treating the output matrix as an image, in which case you should set "`graphical: true`".

In the example given, the nameless function inside `gpu.createKernel(functionDef,options)` provides this mechanism. The function definition `functionDef` looks like Javascript, but instead is a restricted subset of Javascript, limited by the WebGL mechanism used.

The sample program has the following definition of a function, which takes three input arrays (`Camera`, `Lights`, and `Objects`), and computes a single colour value (incorrectly spelt as `color` by GPU.js :).

```
var opt = {
         dimensions: [800,600],
         debug:      true,
         constants:  {OBJCOUNT: objects[0], SPHERE: ObjTyp.SPHERE},
         graphical:  true,
         mode:       mode
      };
var y = gpu.createKernel(function(Camera,Lights,Objects) {
         var idx = 1;                                    // index for looking through all the objects
         var nextidx = 1;
         this.color(0.95,0.95,0.95);                     // By default canvas is light grey
         for (i=0; i<this.constants.OBJCOUNT; i++ ) {    // Look at all object records
             idx = nextidx;                              // Skip to next record
             nextidx = Objects[idx+1]+idx;               // Pre-compute the beginning of the next record
             if (Objects[idx] == this.constants.SPHERE) { // i.e. if it is a SPHERE...
                if (dist(this.thread.x,this.thread.y,Objects[idx+9],Objects[idx+10]) < Objects[idx+12]) {
                    this.color(Objects[idx+2],Objects[idx+3],Objects[idx+4]);
                }
             }
         }
      }, opt);
```

In this (2D) example, the inner loop is iterating over the number of objects in the `Objects` array. An index `idx` jumps through the array looking at each object record. The first field in each record identifies the type of the record, the second field is the size of this record. In the case that the current object is a `SPHERE` (`=0.0`) object, the sample program computes the x-y distance between this pixel and the x-y centre of the sphere, and if it is less than the radius of the sphere, it colours the pixel according to the object colour. Note that the function only computes one value, the value for this element (pixel) on the 2D matrix.

Note also that in your program, you will be doing entirely different things in this function, computing rays, checking for intersections, iterating over reflections, and so on. You may define multiple kernels, doing work in some sort of pipeline. For example, you might be computing values in a matrix, in which case your option for that kernel should be set "`graphical:` false", and it should specifically return a value:

```
var mat_mult = gpu.createKernel(function(A, B) {
    var sum = 0;
    for (var i=0; i<512; i++) {
       sum += A[this.thread.y][i] * B[i][this.thread.x];
    }
    return sum;
}, opt);
```

1.1 Sample input formats

It is common in 3D scene construction to specify elements of the scene in three or four areas.

A *camera* (or viewpoint) indicates the location of the camera or eye viewing the scene, the direction in which the camera is pointed, and a field of view. In our case, we can only specify this as an array of `float32` values, so we will use this:

```
   var camera = [
      0,20,-200,                // x,y,z coordinates
      0,0,1,                    // Direction normal vector
      45                        // Horizontal field of view : example 45
   ];
```

Note that if you change the values of the elements of `camera[]` at runtime, then the display could change. It might be good for your program to provide a way of changing `camera[]`.

A *light* array indicates the location, and colour of light sources in the scene. In our case, we will use this:

```
var lights = [
  2,                        // number of lights
  200,200,200, 0,1,0,       // light 1, x,y,z location, and rgb colour (green)
  100,100,100, 1,1,1,       // light 2, x,y,z location, and rgb colour (white)
];
```

Note that if you change the values of the elements of the `lights[]` array at runtime, then the display could change. It might be good for your program to provide a way of changing `lights[]`.

An *objects* array indicates the type, colour, material, location, and other characteristics of simple solid objects in the scene. In our case, we will use this format, with each line followed by a comment field:

```
var objects = [
    5,
 // number of objects
    SPHERE,      13, 1.0,0.0,0.0,  0.2, 0.7,0.1, 1.0, 10,50,50,  40,
 //    typ,   recsz,   r,  g,  b, spec,lamb,amb,opac,  x, y, z, rad,
    CUBOID,      18, 1.0,0.0,0.0,  0.2, 0.7,0.1, 1.0, 10,20,40,  40, 20, 10,  1, 1, 1,
 //    typ,   recsz,   r,  g,  b, spec,lamb,amb,opac,  x, y, z, wid,hgt,dep, xd,yd,zd,
    CYLINDER,    17, 1.0,0.0,0.0,  0.2, 0.7,0.1, 1.0, 10,20,30,  40, 10,      1, 1, 1,
 //      typ, recsz,   r,  g,  b, spec,lamb,amb,opac,  x, y, z, rad,hgt,     xd,yd,zd,
    CONE,        17, 1.0,0.0,0.0,  0.2, 0.7,0.1, 1.0, 10,20,30,  40, 10,      1, 1, 1,
 //  typ,     recsz,   r,  g,  b, spec,lamb,amb,opac,  x, y, z, rad,hgt,     xd,yd,zd,
    TRIANGLE,    19, 1.0,0.0,0.0,  0.2, 0.7,0.1, 1.0, 10,20,30, 10,20,40, 40,50,60,     2,
 //      typ, recsz,   r,  g,  b, spec,lamb,amb,opac, x1,y1,z1, x2,y2,z2, x3,y3,z3, width,
];
```

where SPHERE, CUBOID... are actually the numbers 1,2,3,4,5. The TRIANGLE shape is defined by its vertices, and has width `width`. The other terms used in the comment field are given below:

Name and brief description			
`typ`	shape/type of solid object	`recsz`	number of elements in record
`r,g,b`	red green and blue values	`x,y,z`	coordinates of centre
`spec`	(0-1) for specular reflection	`lamb`	(0-1) Lambertian model reflection
`amb`	(0-1) ambient color	`opac`	(0-1) Opacity of object
`wid`	Size in x direction	`hgt`	Size in y direction
`dep`	Size in z direction	`xd,yd,zd`	Normal direction vector from origin
`rad`	Radius	`xn,yn,zn`	xyz coordinates of specific vertices

To orient yourself for each object, imagine the x,y plane being left to right and up and down on your screen, with the z axis going into the screen. Each object is placed on a flat horizontal table (the xz plane). So a CYLINDER or CONE has the circle base on the plane, and the long axis going upwards (y). The xyz values determine the origin of the shape, in the case of the SPHERE, the centre of the sphere, in the case of the CYLINDER or CONE it is the centre of the circle base, and in the case of the CUBOID it is the centre of gravity of the element. The xd,yd,zd direction vector gives the direction in which the principal (y) axis is pointing.

1.2 Properties of GPU.js

Documentation for GPU.js may be found at

`http://gpu.rocks/getting-started/`

GPU.js was developed at NUS during Hack&Roll this year, and so there is a fair amount of local knowledge about it. To be more precise... all knowledge about it is local! The source is freely available from the website.

The subset of Javascript used by GPU.js has some restrictions that are a little confusing, beginning with the restriction that the data types in the kernel functions can only be `float32` (a restriction imposed by the OpenGL shader architecture). In the example, three `float32` arrays are fed to the `functionDef`, and the output is a 2D `float32` matrix, which we treat as an rgba image. Loop bounds must be fixed at compile time, so the `constants` option helps you import constant values into your GPU function. The example program shows how to do this. There is no recursion. (!) ... GPUs generally do not do recursion.

The functions you can call within a kernel, can only be ones written in the restricted subset allowed by GPU.js, and a mechanism is provided to attach these functions to the kernel, to help you structure your program. For example, the example program calls a function `sqr(x)`, implemented in this way:

```
function sqr(x) {
   return x*x;
}
gpu.addFunction(sqr);
```

1.3 Completion? Accuracy?

There are various references to *completion* in the project specification just given. In your project paper presentation, you should clearly state at the beginning which components of the problem you implemented. There is an almost endless list of possible things to do, or not do:

- Did you do all the object types? Did you implement lights and hence shadows? Did you implement the camera, or do you just have a fixed viewpoint?
- Did you support all modes of reflection (using the ambient/diffuse/specular fields)? What type/level of raytracing did you implement? How many bounces?
- Did you do anything special for accuracy? Speedup? Workload reduction?
- Did you use any other special techniques to improve the system? (For example: antialiasing).
- Can you adjust the parameters to specify different methods of operation? Did you animate your scene? Did you provide some sort of camera motion/fly-through technique?
- Did you provide sample scenes that provide evidence of testing?

There is also mention of accuracy. Since GPU.js only deals with `float32` values, it may seem that this defines the *accuracy* of your implementation. It is possible that the order of operations may have an affect on *this* type of accuracy.

However, there is also the accuracy of the raytracing itself. If, for example, you only consider a single light bounce, then the *accuracy* may be limited by this.

1.4 Ray tracing

There are many ways you can learn about ray tracing. If you do not know where to start, I suggest you look at Wikipedia, and perhaps this page:

`http://www.macwright.org/literate-raytracer/`

This page describes a very simple ray tracer, in Javascript. It is written in a literate-programming style, and is quite understandable. We will also try to help with this aspect in class or tutorials. You may also find other WebGL ray tracers, that work using on-the-fly compilation of shaders embedded in script tags, such as this one:

`https://rawgit.com/sschoenholz/WebGL-Raytracer/master/index.html`

You can use them for ideas, but you are NOT to use this GLSL shader technique. All your code must be GPU.js code; there should be no use of `createShaderFromScriptElement()`.

2 Part 2

In part 2, we want you to write a short paper, containing a brief outline of your implementation (what you did, or did not do), showing the (tabulated) results you got from your implementation, evaluating, analysing and reflecting on why the results are the way they are.

Note that there are lots of variables in this evaluation. It is up to you to choose which components that you wish to address. For example, when evaluating problem size you might consider the screen size, or number of objects in a scene, or even the number of reflections you consider. When evaluating performance you might consider measures like FPS, or perhaps workload reduction (if you come up with some strategy to reduce the computations needed). We expect to see at least the following items:

- Implementation: Outline of what you achieved in your implementation.
- Results: Any tabulated results.
- Hardware: Discussions on the hardware you used, and how the hardware contributes to the results obtained.
- Speedup/Accuracy: A discussion on what is causing the behaviour you record.

In addition, we want you to reflect on the techniques you used/developed to improve the speed, workload reduction, or accuracy of your implementation. In your analysis can you demonstrate that your technique(s) was/were effective?

3 Assessment

The assessment below is only a guideline, and is indicative of the project assessment. However, the assessment for individual projects may deviate from this in some ways, dependent on the form of the delivered project. On **Friday 15th April 17:00**, the project is due. It will be worth **30/100** of your final mark. The assessment will be done as follows:

- **10/30** Implementation: An assessment of the quality of the implementation you present, and the techniques used to improve the behaviour.
- **5/30** Completion: These marks are for an assessment of the level of completion of your project.
- **10/30** Reflection: An assessment of your writeup in Part 2, with respect to your analysis of the hardware effects, and analysis of the techniques used to improve the behaviour.
- **5/30** Extra effort: These marks are reserved for those projects which clearly reflect extra effort.

In general, better assessments will be given to implementations which encapsulate more complex ideas.

3.1 Finally...

Your final completed project should be submitted to the Project2 IVLE workbin in the form of a single zipped up file name A00XXXXXX.zip (i.e. your-student-number.zip), containing your writeup, commented sources and any other supporting files, on or before Friday 15th April 17:00. You must also submit a readable **hardcopy** to your lecturer.

You are allowed to discuss the problems with your friends, and to study any background material with them, but the project *should be your own work*. **Copying** and **cheating** will be grounds for failing the project.