# Problem 4: CUDA

**20 points.** In Assignment 2, we had you implement a Cuda rasterizer for circles as quickly as you could using primitive colors. Most modern day images ditch these primitive colors for more advanced textures, requiring expensive memory look-ups for these texture colors in favor of more realistic visual results. In this problem, we will be extending the Cuda rasterizer to work with texture look-ups while still aiming for fast parallel performance.

One assumption we will be making here is that *all primitives are fully opaque*. Thus, you do not have to worry about blending colors across circles (like in Assignment 2), rather you only need to focus on rendering the primitive closest to the camera (i.e., the triangle with the smallest z value). We would like to write a rasterization pipeline in Cuda that takes advantage of this assumption to make the pixel shading as fast as possible. Below we have provided you with starter code for our rasterizer.

We have provided you with 3 black-box helper functions listed below, similar to Assignment 2:

---

```
// returns true if pixelNorm lies in the triangle with vertices (a,b,c)
__global__ bool in_primitive(float3 a, float3 b, float3 c,
                             float2 pixelNorm);

// rasterizes the triangle using pInfo->texture
__global__ void rasterize(float2 pixelNorm, primitiveInfo* pInfo);

// returns interpolated depth of the triangle at pixelNorm
__global__ float depth_at_point(float3 a, float3 b, float3 c,
                                float2 pixelNorm);
```

---

On the next page is an implementation of our rasterizer that works with texture look-ups. You may use our implementation as starter code for any code that you are asked to write in this problem.

```
__global__ void kernelRenderPrimitives() {

  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int numThreads = blockDim.x * numBlocks.x;
  int width = cuConstRendererParams.screenWidth;
  int height = cuConstRendererParams.screenHeight;

  float invWidth = 1.f / cuConstRendererParams.screenWidth;
  float invHeight = 1.f / cuConstRendererParams.screenHeight;

  // this is a global variable
  float bestDepth[height, width];

  int numPrimitives = cuConstRendererParams.numPrimitives;
  for (int i = index; i < numPrimitives; i += numThreads) {

    // get primitive vertices
    float3 a = *(float3*)(&cuConstRendererParams.position[3 * (3 * i + 0)]);
    float3 b = *(float3*)(&cuConstRendererParams.position[3 * (3 * i + 1)]);
    float3 c = *(float3*)(&cuConstRendererParams.position[3 * (3 * i + 2)]);

    // pinfo->texture stores the texture as we as other primitive data
    pInfo *pinfo = (pInfo*)(&cuConstRendererParams.primitiveInfo[i]);

    // defines primitive bounding-box in 2D using orthographic projection
    // clip into [0,0] - [wth,hgt] range
    int minX = max(min(min(a.x, b.x), c.x), 0);
    int maxX = min(max(max(a.x, b.x), c.x), width);
    int minY = max(min(min(a.y, b.y), c.y), 0);
    int maxY = min(max(max(a.y, b.y), c.y), height);


    for (int pixelX = minX; pixelX <= maxX; pixelX++) {
      for (int pixelY = minY; pixelY <= maxY; pixelY++) {
        // compute normalized pixel coordinates
        float2 pixelNorm = make_float2(invWidth * (float(pixelX) + 0.5f),
                                       invHeight * (float(pixelY) + 0.5f));
        if ( in_primitive(a, b, c, pixelNorm) ) {
          // rasterizes primitive using the texture.
          // TODO: is this correct?
          float d = depth_at_point(a, b, c, pixelNorm);
          if(d < bestDepth[pixelY, pixelX]) {
             Lock { // low-level mutex code elided ...
               rasterize(pixelNorm, pinfo);
               bestDepth[pixelY, pixelX] = d;
             }
          }
        }
      }
    }
  }
}
```

## 4.1) Correctness (5 points)

Your 418 partner isn't sure as to whether a specific part of the code is correct and marks it with a `TODO` comment. You run the results and indeed, there is a bug with your code affecting the visual output. What is the bug? How would you go about fixing the problem?

## 4.2) Performance (5 points)

You find the above code to be very slow, so you try a different avenue of parallelism where you parallelize over pixels instead (again, similar to Assignment 2). For each pixel, you iterate through all the primitives and hold onto a reference of the closest primitive before rasterizing it in the end, minimizing the amount of calls to rasterize to just one per thread.

```
for each pixel: ← (parallel over threads)
     for each primitive: //we are looping through all primitives
          if depth(pixel, primitive) < bestDepth:
               bestPrimitive = primitive;
               bestDepth = depth(pixel, primitive);
     rasterize(bestPrimitive);
```

Does this new implementation require any synchronization? Please refer to whether memory operations are independent/shared among threads when justifying your answer.

## 4.3) Work balance (4 points)

Compare the original implementation to the modified implementation in 4.2. For each implementation, is there any workload imbalance amongst the threads in the same block? What about threads in between blocks? You may assume the call to rasterize takes the same amount of time for each primitive.

For the below questions, assume you are running your revised kernel on a 6 core GPU with 64 warps and 32 threads each. The device has 64kB of available shared memory total per device.

## 4.4) Block size (3 points)

You try running your revised kernel with a thread block size of (64,64,1), but you receive a CUDA error. You run it again on a block size of (32,32,1) and your program works fine. Why might that be the case?

## 4.5) Shared memory (3 points)

Running your revised kernel requires 16kB of shared memory per block with a block size of (32,32,1). Do you expect cores to have full occupancy (percentage of cores in use)? Why or why not?