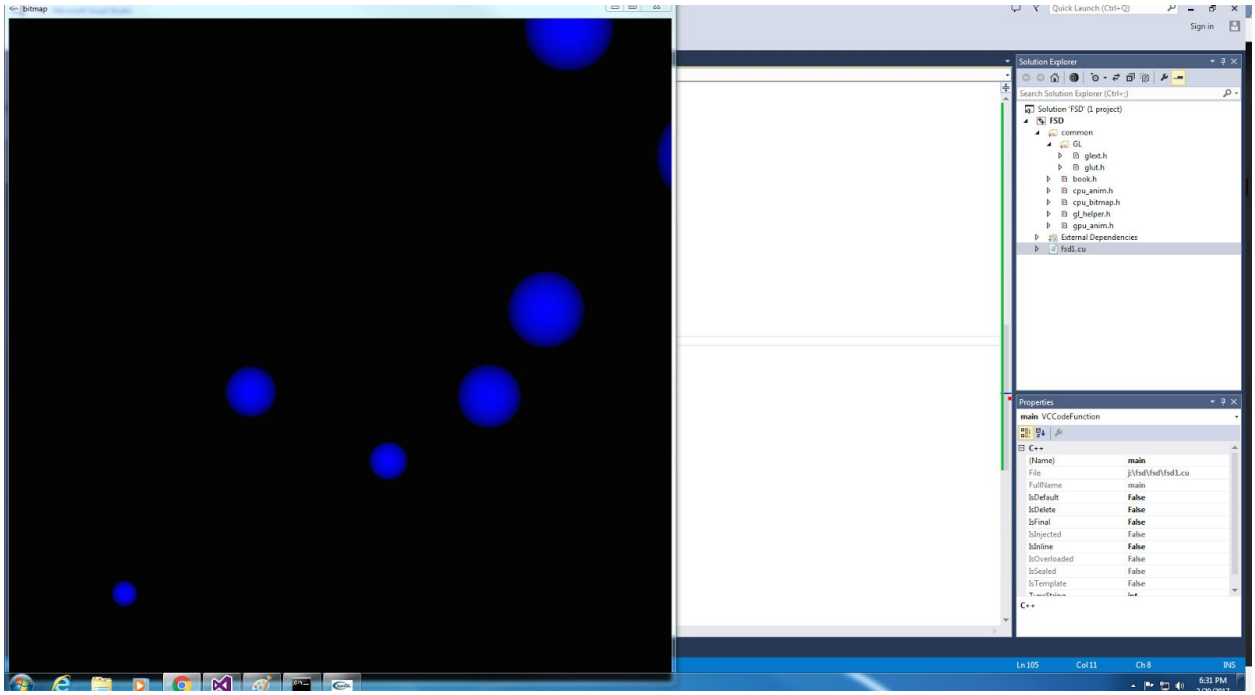
**2 a. Which variable(s) control the number of spheres? **
Answer :SPHERES

2 b. Which variable(s) control the color(s) of the spheres?
Answer: r,g,b

2 c. Which variable(s) control the location of the spheres?
Answer: x,y,z

2 d. Which variable(s) control the size of each sphere?
Answer: radius

3) a)



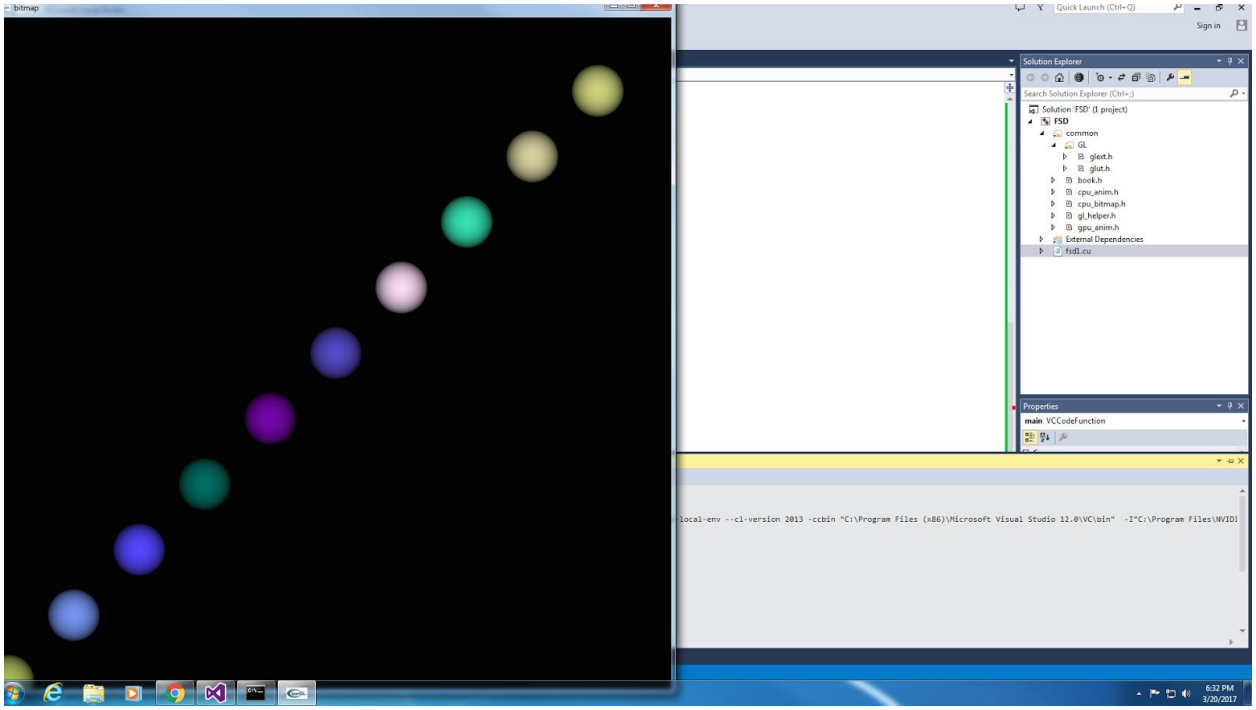

```
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include "J:\ami\ami/common/book.h"
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>
#include "..\..\ami\ami\common\cpu_bitmap.h"
```

```
#include<math.h>

#define DIM 1024

#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f

struct Sphere {
        float   r, b, g;
        float   radius;
        float   x, y, z;
        __device__ float hit(float ox, float oy, float *n) {
                float dx = ox - x;
                float dy = oy - y;
                if (dx*dx + dy*dy < radius*radius) {
                        float dz = sqrtf(radius*radius - dx*dx - dy*dy);
                        *n = dz / sqrtf(radius * radius);
                        return dz + z;
                }
                return -INF;
        }
};
#define SPHERES 10

__constant__ Sphere s[SPHERES];

__global__ void kernel(unsigned char *ptr) {
        // map from threadIdx/BlockIdx to pixel position
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int offset = x + y * blockDim.x * gridDim.x;
        float   ox = (x - DIM / 2);
        float   oy = (y - DIM / 2);

        float   r = 0, g = 0, b = 0;
        float   maxz = -INF;
        for (int i = 0; i<SPHERES; i++) {
                float   n;
                float   t = s[i].hit(ox, oy, &n);
                if (t > maxz) {
                        float fscale = n;
                        r = s[i].r * fscale;
                        g = s[i].g * fscale;
```

```
                        b = s[i].b * fscale;
                        maxz = t;
                }
        }

        ptr[offset * 4 + 0] = (int)(r * 255);
        ptr[offset * 4 + 1] = (int)(g * 255);
        ptr[offset * 4 + 2] = (int)(b * 255);
        ptr[offset * 4 + 3] = 255;
}

// globals needed by the update routine
struct DataBlock {
        unsigned char   *dev_bitmap;
};

int main(void) {
        DataBlock   data;
        // seed the random function
        srand(time(NULL));
        // capture the start time
        cudaEvent_t     start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start, 0);

        CPUBitmap bitmap(DIM, DIM, &data);
        unsigned char   *dev_bitmap;

        // allocate memory on the GPU for the output bitmap
        cudaMalloc((void**)&dev_bitmap,
                bitmap.image_size());

        // allocate temp memory, initialize it, copy to constant
        // memory on the GPU, then free our temp memory
        Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere)* SPHERES);
        for (int i = 0; i<SPHERES; i++) {
                temp_s[i].r = 0;
                temp_s[i].g = 0;
                temp_s[i].b = 1;
                temp_s[i].x = rnd(i * 100 - 500);
                temp_s[i].y = rnd(i * 100 - 500);
                temp_s[i].z = rnd(i * 100 - 500);
```

```
            temp_s[i].radius = i * 10 + 20;
        }
        cudaMemcpyToSymbol(s, temp_s,
            sizeof(Sphere)* SPHERES);
        free(temp_s);

        // generate a bitmap from our sphere data
        dim3    grids(DIM / 16, DIM / 16);
        dim3    threads(16, 16);
        kernel << <grids, threads >> >(dev_bitmap);

        // copy our bitmap back from the GPU for display
        cudaMemcpy(bitmap.get_ptr(), dev_bitmap,
            bitmap.image_size(),
            cudaMemcpyDeviceToHost);

        // get stop time, and display the timing results
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        float   elapsedTime;
        cudaEventElapsedTime(&elapsedTime,
            start, stop);
        printf("Time to generate:  %3.1f ms\n", elapsedTime);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        cudaFree(dev_bitmap);

        // display
        bitmap.display_and_exit();
}
```

3)b)



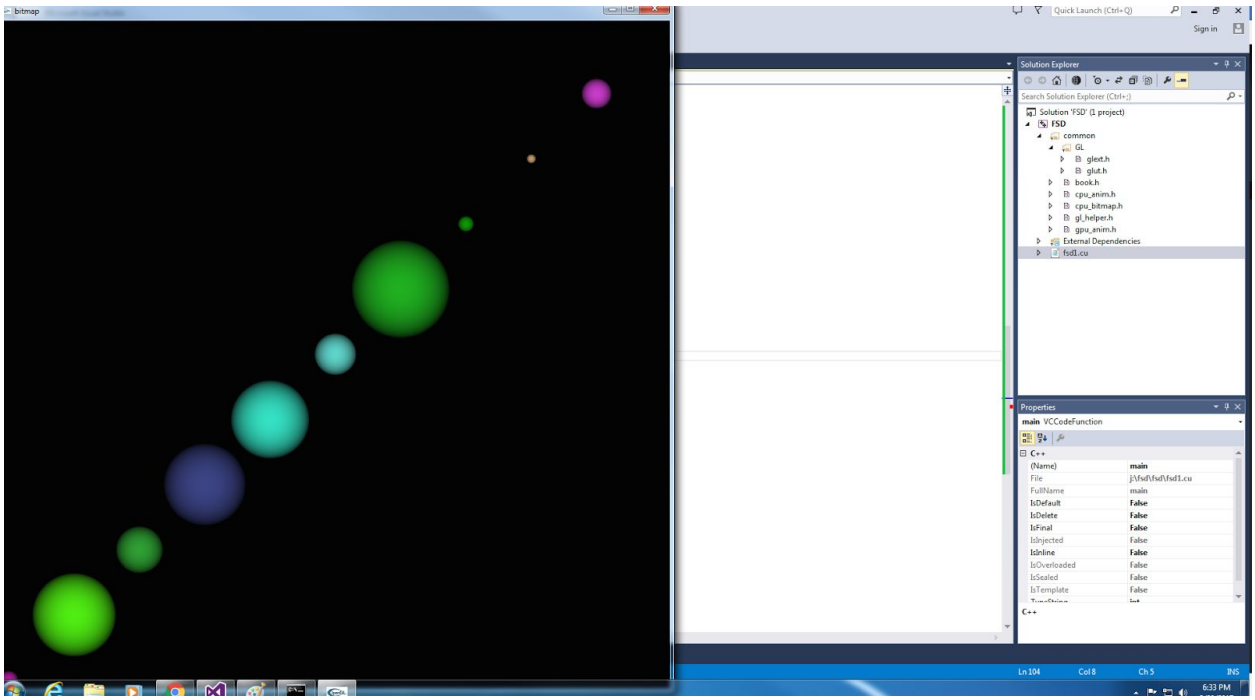

```
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include "J:\ami\ami/common/book.h"
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>
#include "..\..\ami\ami\common\cpu_bitmap.h"
#include<math.h>

#define DIM 1024

#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f

struct Sphere {
        float   r, b, g;
        float   radius;
        float   x, y, z;
        __device__ float hit(float ox, float oy, float *n) {
```

```
                float dx = ox - x;
                float dy = oy - y;
                if (dx*dx + dy*dy < radius*radius) {
                        float dz = sqrtf(radius*radius - dx*dx - dy*dy);
                        *n = dz / sqrtf(radius * radius);
                        return dz + z;
                }
                return -INF;
        }
};
#define SPHERES 10

__constant__ Sphere s[SPHERES];

__global__ void kernel(unsigned char *ptr) {
        // map from threadIdx/BlockIdx to pixel position
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int offset = x + y * blockDim.x * gridDim.x;
        float   ox = (x - DIM / 2);
        float   oy = (y - DIM / 2);

        float   r = 0, g = 0, b = 0;
        float   maxz = -INF;
        for (int i = 0; i<SPHERES; i++) {
                float   n;
                float   t = s[i].hit(ox, oy, &n);
                if (t > maxz) {
                        float fscale = n;
                        r = s[i].r * fscale;
                        g = s[i].g * fscale;
                        b = s[i].b * fscale;
                        maxz = t;
                }
        }

        ptr[offset * 4 + 0] = (int)(r * 255);
        ptr[offset * 4 + 1] = (int)(g * 255);
        ptr[offset * 4 + 2] = (int)(b * 255);
        ptr[offset * 4 + 3] = 255;
}

// globals needed by the update routine
```

```
struct DataBlock {
	unsigned char   *dev_bitmap;
};

int main(void) {
	DataBlock   data;
	// seed the random function
	srand(time(NULL));
	// capture the start time
	cudaEvent_t     start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);
	cudaEventRecord(start, 0);

	CPUBitmap bitmap(DIM, DIM, &data);
	unsigned char   *dev_bitmap;

	// allocate memory on the GPU for the output bitmap
	cudaMalloc((void**)&dev_bitmap,
		bitmap.image_size());

	// allocate temp memory, initialize it, copy to constant
	// memory on the GPU, then free our temp memory
	Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere)* SPHERES);
	for (int i = 0; i<SPHERES; i++) {
		temp_s[i].r = rnd(1.0f);
		temp_s[i].g = rnd(1.0f);
		temp_s[i].b = rnd(1.0f);
		temp_s[i].x = i * 100 - 500;
		temp_s[i].y = i * 100 - 500;
		temp_s[i].z = i * 100 - 500;
		temp_s[i].radius = 40;
	}
	cudaMemcpyToSymbol(s, temp_s,
		sizeof(Sphere)* SPHERES);
	free(temp_s);

	// generate a bitmap from our sphere data
	dim3    grids(DIM / 16, DIM / 16);
	dim3    threads(16, 16);
	kernel << <grids, threads >> >(dev_bitmap);

	// copy our bitmap back from the GPU for display
```

```
        cudaMemcpy(bitmap.get_ptr(), dev_bitmap,
                bitmap.image_size(),
                cudaMemcpyDeviceToHost);

        // get stop time, and display the timing results
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        float   elapsedTime;
        cudaEventElapsedTime(&elapsedTime,
                start, stop);
        printf("Time to generate:  %3.1f ms\n", elapsedTime);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        cudaFree(dev_bitmap);

        // display
        bitmap.display_and_exit();
}
```

3) c)



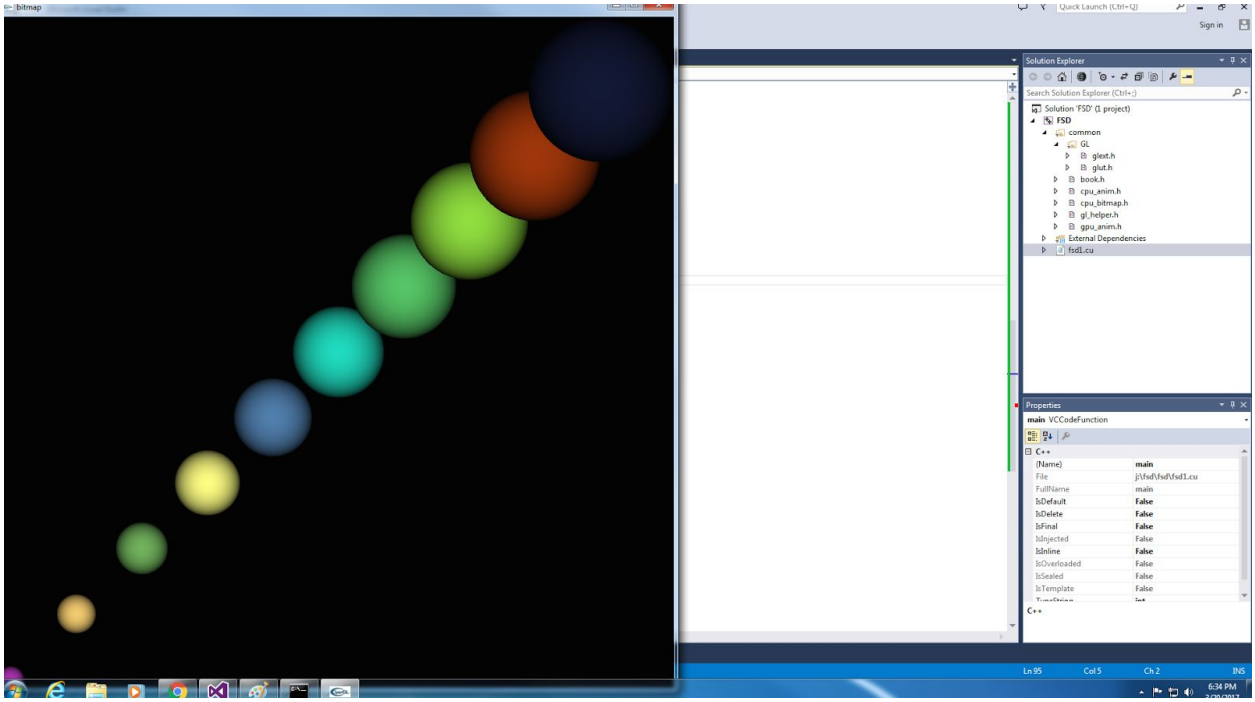

```
#include <cuda_runtime_api.h>
```

```
#include "device_launch_parameters.h"
#include "J:\ami\ami/common/book.h"
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>
#include "..\..\ami\ami\common\cpu_bitmap.h"
#include<math.h>

#define DIM 1024

#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f

struct Sphere {
        float   r, b, g;
        float   radius;
        float   x, y, z;
        __device__ float hit(float ox, float oy, float *n) {
                float dx = ox - x;
                float dy = oy - y;
                if (dx*dx + dy*dy < radius*radius) {
                        float dz = sqrtf(radius*radius - dx*dx - dy*dy);
                        *n = dz / sqrtf(radius * radius);
                        return dz + z;
                }
                return -INF;
        }
};
#define SPHERES 10

__constant__ Sphere s[SPHERES];

__global__ void kernel(unsigned char *ptr) {
        // map from threadIdx/BlockIdx to pixel position
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int offset = x + y * blockDim.x * gridDim.x;
        float   ox = (x - DIM / 2);
        float   oy = (y - DIM / 2);
```

```
        float   r = 0, g = 0, b = 0;
        float   maxz = -INF;
        for (int i = 0; i<SPHERES; i++) {
                float   n;
                float   t = s[i].hit(ox, oy, &n);
                if (t > maxz) {
                        float fscale = n;
                        r = s[i].r * fscale;
                        g = s[i].g * fscale;
                        b = s[i].b * fscale;
                        maxz = t;
                }
        }

        ptr[offset * 4 + 0] = (int)(r * 255);
        ptr[offset * 4 + 1] = (int)(g * 255);
        ptr[offset * 4 + 2] = (int)(b * 255);
        ptr[offset * 4 + 3] = 255;
}

// globals needed by the update routine
struct DataBlock {
        unsigned char   *dev_bitmap;
};

int main(void) {
        DataBlock   data;
        // seed the random function
        srand(time(NULL));
        // capture the start time
        cudaEvent_t     start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start, 0);

        CPUBitmap bitmap(DIM, DIM, &data);
        unsigned char   *dev_bitmap;

        // allocate memory on the GPU for the output bitmap
        cudaMalloc((void**)&dev_bitmap,
                bitmap.image_size());

        // allocate temp memory, initialize it, copy to constant
```

```
        // memory on the GPU, then free our temp memory
        Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere)* SPHERES);
        for (int i = 0; i<SPHERES; i++) {
                temp_s[i].r = rnd(1.0f);
                temp_s[i].g = rnd(1.0f);
                temp_s[i].b = rnd(1.0f);
                temp_s[i].x = i * 100 - 500;
                temp_s[i].y = i * 100 - 500;
                temp_s[i].z = i * 100 - 500;
                temp_s[i].radius = rnd(80);
        }
        cudaMemcpyToSymbol(s, temp_s,
                sizeof(Sphere)* SPHERES);
        free(temp_s);

        // generate a bitmap from our sphere data
        dim3    grids(DIM / 16, DIM / 16);
        dim3    threads(16, 16);
        kernel << <grids, threads >> >(dev_bitmap);

        // copy our bitmap back from the GPU for display
        cudaMemcpy(bitmap.get_ptr(), dev_bitmap,
                bitmap.image_size(),
                cudaMemcpyDeviceToHost);

        // get stop time, and display the timing results
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        float   elapsedTime;
        cudaEventElapsedTime(&elapsedTime,
                start, stop);
        printf("Time to generate:  %3.1f ms\n", elapsedTime);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        cudaFree(dev_bitmap);

        // display
        bitmap.display_and_exit();
}
```

3)d)




```
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include "J:\ami\ami/common/book.h"
#include <cuda_runtime_api.h>
#include "device_launch_parameters.h"
#include <stdio.h>
#include <cuda.h>
#include <stdlib.h>
#include <time.h>
#include "..\..\ami\ami\common\cpu_bitmap.h"
#include<math.h>

#define DIM 1024

#define rnd( x ) (x * rand() / RAND_MAX)
#define INF     2e10f

struct Sphere {
        float   r, b, g;
        float   radius;
        float   x, y, z;
        __device__ float hit(float ox, float oy, float *n) {
```

```
                float dx = ox - x;
                float dy = oy - y;
                if (dx*dx + dy*dy < radius*radius) {
                        float dz = sqrtf(radius*radius - dx*dx - dy*dy);
                        *n = dz / sqrtf(radius * radius);
                        return dz + z;
                }
                return -INF;
        }
};
#define SPHERES 10

__constant__ Sphere s[SPHERES];

__global__ void kernel(unsigned char *ptr) {
        // map from threadIdx/BlockIdx to pixel position
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int offset = x + y * blockDim.x * gridDim.x;
        float   ox = (x - DIM / 2);
        float   oy = (y - DIM / 2);

        float   r = 0, g = 0, b = 0;
        float   maxz = -INF;
        for (int i = 0; i<SPHERES; i++) {
                float   n;
                float   t = s[i].hit(ox, oy, &n);
                if (t > maxz) {
                        float fscale = n;
                        r = s[i].r * fscale;
                        g = s[i].g * fscale;
                        b = s[i].b * fscale;
                        maxz = t;
                }
        }

        ptr[offset * 4 + 0] = (int)(r * 255);
        ptr[offset * 4 + 1] = (int)(g * 255);
        ptr[offset * 4 + 2] = (int)(b * 255);
        ptr[offset * 4 + 3] = 255;
}

// globals needed by the update routine
```

```
struct DataBlock {
	unsigned char   *dev_bitmap;
};

int main(void) {
	DataBlock   data;
	// seed the random function
	srand(time(NULL));
	// capture the start time
	cudaEvent_t     start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);
	cudaEventRecord(start, 0);

	CPUBitmap bitmap(DIM, DIM, &data);
	unsigned char   *dev_bitmap;

	// allocate memory on the GPU for the output bitmap
	cudaMalloc((void**)&dev_bitmap,
		bitmap.image_size());

	// allocate temp memory, initialize it, copy to constant
	// memory on the GPU, then free our temp memory
	Sphere *temp_s = (Sphere*)malloc(sizeof(Sphere)* SPHERES);
	for (int i = 0; i<SPHERES; i++) {
		temp_s[i].r = rnd(1.0f);
		temp_s[i].g = rnd(1.0f);
		temp_s[i].b = rnd(1.0f);
		temp_s[i].x = i * 100 - 500;
		temp_s[i].y = i * 100 - 500;
		temp_s[i].z = i * 100 - 500;
		temp_s[i].radius = i * 10 + 20;
	}
	cudaMemcpyToSymbol(s, temp_s,
		sizeof(Sphere)* SPHERES);
	free(temp_s);

	// generate a bitmap from our sphere data
	dim3    grids(DIM / 16, DIM / 16);
	dim3    threads(16, 16);
	kernel << <grids, threads >> >(dev_bitmap);

	// copy our bitmap back from the GPU for display
```

```
        cudaMemcpy(bitmap.get_ptr(), dev_bitmap,
                bitmap.image_size(),
                cudaMemcpyDeviceToHost);

        // get stop time, and display the timing results
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        float   elapsedTime;
        cudaEventElapsedTime(&elapsedTime,
                start, stop);
        printf("Time to generate:  %3.1f ms\n", elapsedTime);

        cudaEventDestroy(start);
        cudaEventDestroy(stop);

        cudaFree(dev_bitmap);

        // display
        bitmap.display_and_exit();
}
```