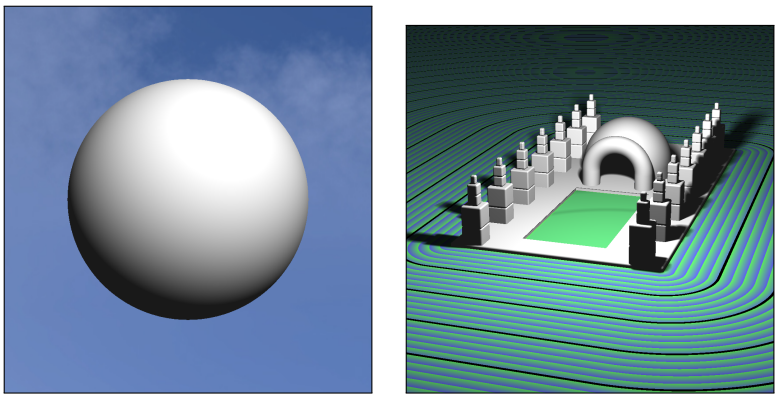
Further Graphics Tick

Ray-Marched Signed Distance Fields



(a) A sphere (b) A building created with SDF

Figure 1: In this exercise you will use *Signed Distance Fields* to render an interactive ray-cast scene on the GPU.

1 Introduction

In this exercise you will render an interactive image using *Signed Distance Fields*. Given the skeleton of a *ray marcher*, you will write code in GLSL, the OpenGL Shader Language, and your code will run in a Java OpenGL framework designed to capture images from preset points of view for marking.

The key idea of a ray marcher is that the color of each pixel is defined by the ray from the camera eye through that pixel. We step along the ray at discrete intervals, until the ray hits an object. The classical lighting equations determine the pixel's color from the ray / object intersection.

The key idea of Signed Distance Fields is that the answer to the questions, "does this ray hit this object?" and "how far can I safely advance along this ray without worrying that I've missed hitting an object?" can be the same. As we fire point p along ray r into the scene, the minimum of the distances from p to all the objects in the scene is the distance along r which p can safely move; and if a distance is zero, the ray has hit an object.

Signed Distance Fields, also sometimes called *sphere tracing*, were brilliantly written up by John C. Hart in his 1996 paper, "Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces", available at `https://graphics.cs.illinois.edu/papers/zeno`.

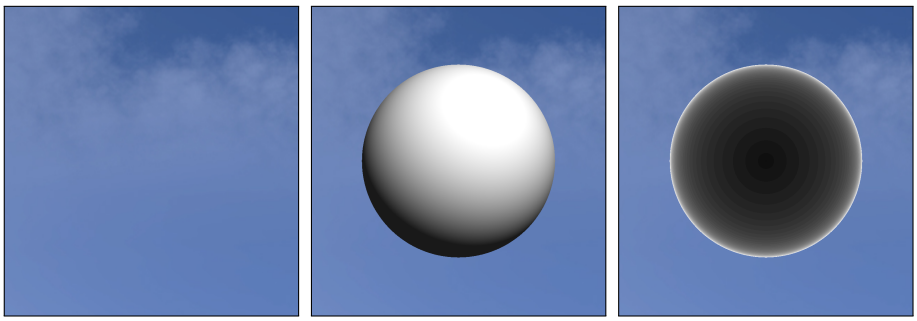


Figure 2: If you have set the tick up correctly, you will see a cloudy blue sky. Use the mouse to spin your point of view (but there's not much to see). If you uncomment the line which calls `sphere()` in the `raymarch()` method of `fragment_shader.glsl`, a sphere should appear. Press R to visualize the number of raymarching steps per pixel of the sphere (black = min, white = max).

2 Getting started

The practical materials are available online: `https://www.vle.cam.ac.uk/course/view.php?id=145061&sectionid=2027981` / Starter code. It contains the following files and directories:

- tick
  - lib
    - JOML — *Java math library for OpenGL calculations*
    - LWJGL — *Lightweight Java Game Library*
  - resources
    - fragment_shader.glsl — *You will write your ray-marching renderer here*
    - sky.jpg — *Background texture*
    - vertex_shader.glsl — *Processes vertex data*
  - src/uk/ac/cam/cl/furthergfx/crsid/tick
    - Camera.java — *Controls 3D camera position*
    - Texture.java — *Utility to load texture data into OpenGL*
    - Tick.java — *Main class*
    - TickApp.java — *Quad geometry, camera, mouse, and keyboard*
    - TickCanvas.java — *Base class - builds and shows a rendering canvas*

You will modify and submit the file highlighted above and a set of screenshots captured from preset points of view.

Follow the instructions in the setup documentation to compile and run the source code for this exercise.

When you first run the app, an OpenGL window will open and you will see a cloudy blue sky. Using the mouse, you can spin your point of view and look at the clouds texture map from other angles. If you uncomment the line

```
// d = sphere(camPos + t * rayDir);
```

of `fragment_shader.glsl` and save the file with the app still running, the app should pick up your changed fragment shader and automatically reload, rendering the white sphere in Figure 2. If you modify `fragment_shader.glsl` and introduce code which is not proper GLSL, an error will be printed to the Java console.

Class `TickApp.java` sets up the following hotkeys for your use:

- 1 Move camera to the positive Z axis
- 2 Move camera to the positive Y axis
- 3 Move camera to an angled view
- R Toggle between visualization of realistic shading vs the number of ray-marched steps per pixel
- S Capture a screenshot for submission (see below)

2.1 What you'll submit

2.1.1 Screenshots

You will submit a set of screenshots to prove the correctness of your work.

Reference camera positions for each screenshot are hardcoded into `TickApp.java`. When you press S, you will be prompted to select a *Task* from the list below (Figure 3). The camera will move to a preset position and capture a PNG screenshot, named after the Task. Do not rename your screenshot files. This will help demonstrators verify that your outputs are correct.

2.1.2 Task files

Your GLSL code is part of the Tick. You'll be asked to submit the GLSL code for each Task separately, so when a Task is complete, copy its code to a file named "`fragment_shader - Task<n>.glsl`" before you move on.

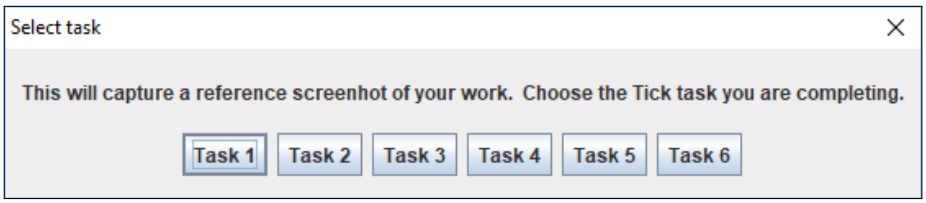


Figure 3: The screenshot picker, which will save an image of your work. Tasks 1-5 are captured from a preset position to ensure consistency.

3 Building and running the Tick

3.1 OpenGL

In this exercise we are using OpenGL to render a simple shape: a quadrilateral which fills the viewport. We then use GLSL, the OpenGL Shading Language, to instruct the GPU to texture the OpenGL viewport with a rendered 3D image.

You will need a computer that can run OpenGL 3.0. Most computers will support OpenGL 3.0, but if your personal machine does not, you can use the MCS machines instead.

We use OpenGL in this assignment because it should be a familiar API and the LWJGL library gives easy-to-encapsulate support for OpenGL to Java. That said, this is not an OpenGL assignment; this is a GLSL assignment.

The OpenGL app we provide is a framework in which your GLSL code is run, but the OpenGL context is not strictly required to run GLSL. A host of WebGL enthusiast websites, such as `shadertoy.com`, offer platforms to host GLSL shaders which you can edit and run directly on the site. If you want to show off your work on a mobile device, consider porting your tick to OpenGL ES for WebGL and uploading it to shadertoy (after you've completed the tick, of course.)

For more information on OpenGL and GLSL, the OpenGL Programming Guide 8th Edition provides a good reference. Make sure to only consult up-to-date documentation.

3.2 Compiling and running the code

The instructions below are for compiling and running the code from the command line. Refer to the document *Working with IDEs* for instructions for an IDE.

When compiling the code, it is necessary to specify the classpath to the libraries we will use. To compile, change the current directory to `tick`, then run:

```
javac -classpath lib/JOML/joml.jar:lib/LWJGL/lwjgl.jar:lib/LWJGL/
    lwjgl-util.jar -d ./out src/uk/ac/cam/cl/furthergfx/crsid/tick
    /*.java
```

The `-d` option specifies where to put the compiled classes. It is good practice not to mix source files with compiled code.

Since the program needs to read a few files from the RESOURCES directory, it must be started from the `tick` directory. Moreover, LWJGL library consists of both JAVA classes and a native library, which needs to be specified at start-up using `-Djava.library.path` argument. You can start the program with the following command:

```
java -classpath lib/JOML/joml.jar:lib/LWJGL/lwjgl.jar:lib/LWJGL/
    lwjgl-util.jar:./out -Djava.library.path=lib/LWJGL/native/
    YOUR_OS uk.ac.cam.cl.furthergfx.crsid.tick.Tick
```

Where you should replace `"YOUR_OS"` with either `windows`, `macosx`, or `linux` depending on your operating system.

On OSX you may need to add the argument `-XstartOnFirstThread`. Another common issue on OSX is that the application window can open out of sight, behind other windows. If this is the case, use Mission Control to find the OpenGL window.

3.3 Debugging a raymarcher in GLSL

GLSL is notoriously difficult to debug, because each pixel computes its value in parallel to every other. This can make it difficult to sample values and run tests. However, a few tips and best practices can make the GLSL debugging experience, if not enjoyable, then at least tolerable:

- Make frequent backups of checkpoints in your work. Online source control repos like Github are ideal, but any system that you can use to retrieve old versions of your code is a good idea.
- If an image doesn't match your expectations, look at your code and ask yourself, "what values would have to be wrong for this to fail?" Then set the color of the pixel itself to those values. Remember that X is Red, Y is Green and Z is Blue.
- When in doubt, simplify. Comment out all but the minimal code and then work your way back. '#if' works well too.
- Remember that GLSL can't throw numerical exceptions so errors like divide-by-zero are swallowed silently.
- Always double-check that you've normalized all the vectors that you think are unit vectors.
- Always double-check that you've clamped values to their expected ranges.

- Some internet sources recommend the project [GLSL-Debugger]. We haven't tried it ourselves and can't say whether it's useful or not, but if you try it please let us know.
- Don't be afraid to ask for help!

4 Assignment

Reference images for all tasks are in Figure 4.

4.1 Task 1

Replace the unit sphere with a **unit cube** (Figure 4a). Code for an SDF cube is available online, in the course notes, or in Appendix A.

As you pivot around the cube, you'll notice that the initial `RENDER_DEPTH` setting of 50 is insufficient and we lose pixels from the backs of the sides of the cube as the sides become oblique towards us. This is because the closer a plane is to parallel to the ray-marched ray, the more steps the ray can advance without drawing quite close enough to the plane to touch it. At only 50 steps, it's easy for a ray that would actually hit the cube to run out of steps before it can make contact.

To solve this problem, Signed Distance Fields offer several clever solutions; but for the purpose of this Tick, we'll simply apply brute force. Raise the value of `RENDER_DEPTH` to 800.

4.2 Task 2

Demonstrate **translation**, **union**, **difference**, **blending** and **intersection** (Figure 4b).

Place four cubes at (-3, 0, -3), (3, 0, -3), (-3, 0, 3), and (3, 0, 3). At +1 X-unit and +1 Z-unit from the center of each cube, place four spheres. The center of each sphere should be at (+1, +0, +1) from the center of its adjacent cube (see Figure 4b and Figure 4c). Model the union, difference, blend and intersection respectively of each cube/sphere pair.

For example, in Figure 4b, note how the front sphere and cube blend smoothly together, whereas the back left pair are a discrete union without blending.

There are many functions available to you to perform a smooth blend between two SDFs; you will find one in the course notes, which has been used with $k = 0.2$ in figure Figure 4b.

4.3 Task 3

Demonstrate **procedural texturing** based on the SDF (Figure 4c).

Add a ground plane beneath the cubes, a horizontal plane at Y = -1. Color the plane, but *not* the cubes, with a regular pattern indicating the distance from that point in the plane to the nearest geometry (excluding the plane itself). This is an example of a *procedural texture*. The effect should be that points in the plane which are equidistant from the cubes and spheres should always be colored the same color, regardless of where in the plane they are.

Your texture will repeat itself over intervals of the signed distance:

- Every 1 unit of distance, your distance texture should shade from light green (`0.4, 1, 0.4`) to light blue (`0.4, 0.4, 1`) - GLSL's `mod()` and `mix()` methods will be helpful here
- Every 5 units of distance, your texture should show a clear black line, 0.25 units wide

Like so:

If your distance visualization doesn't quite match the reference image (Figure 4c) near the corners of your cubes, Figure 5 may be useful. And check out Appendix A for more about what you can learn from visualizing the distance function.

4.4 Task 4

Add a **torus** and **specular lighting** (Figure 4d).

Replace the spheres and cubes with a horizontal torus centered at `vec3(0,3,0)` of major radius 3, minor radius 1.

Add specular lighting to your scene by modifying the method `shade()`. For simplicity, assume that every element in your scene has ambient coefficient 0.1, diffuse coefficient 1.0, specular coefficient 1.0, and 'specular shininess' coefficient 256.

4.5 Task 5

Implement **soft shadows** (Figure 4e).

Tip your torus up onto its side, placing it in the XY plane centered at the origin, perpendicular to the Z axis.

Recall that hard-edged shadows can be computed quite easily with signed distance fields: just as you would in raytracing, you can call your raymarching code along the path towards the light from the point being illuminated. If the signed distance field ever drops to zero or below, you'll know you've hit an obstruction. The benefit of an SDF implementation is that now you can also measure when

you get *close* to an intervening object and use that nearest distance to simulate soft shadows.

The soft shadows used in the reference image were rendered with the sample code given in lecture.

4.6 Task 6

Get in touch with your inner **Frank Lloyd Wright** (Figure 4f).

Apply the techniques you've used here to create a **house or building** using signed distance field geometry. There is no single definition of success here–you can build anything with a door, walls and a roof. You'll show off your final creation during the ticking session.

There are a few requirements:

- You must use **repetition** to create at least one row of repeated geometry
- Your creation must be bounded; it cannot stretch to infinity
- You cannot use just cubes
- You get bragging rights if you add animation or custom colors

5 Submission

Use the built-in screencapture feature (S) to export one image for each task of the tick. The screencapture will automatically position the camera to ease comparison against the reference images for Tasks 1-5; for Task 6, you are free to choose your own orientation and position. Submit these six images and your GLSL code.

Submit your GLSL code as six files, one for each Task, compressed into a single .zip file. Name your GLSL files "`fragment_shader - Task<n>.glsl`".

Your program will be checked in detail during the ticking session, and you will be asked to show off your building (Task 6) and then to make changes to your shader 'on the fly' to demonstrate your understanding of your ray marcher.

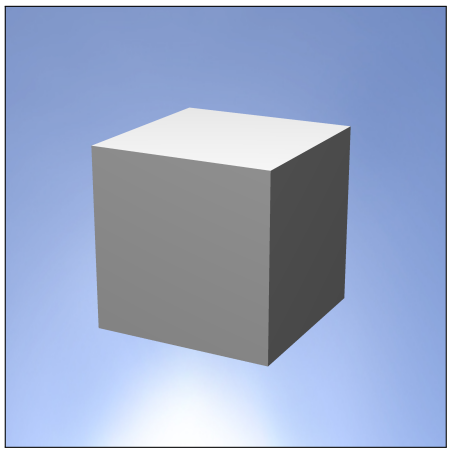
You will need to submit the following files for this tick:

Separately for automated checking:

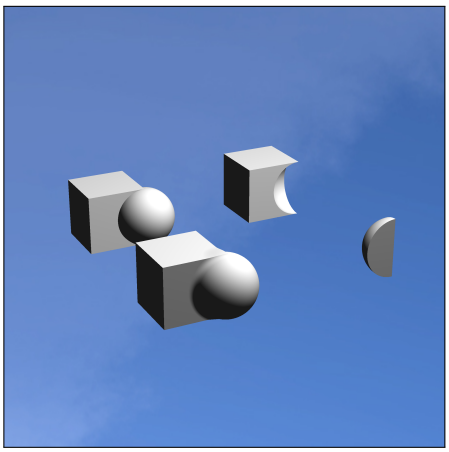
- `Task1.png`
- `Task2.png`
- `Task3.png`
- ...

In a single .zip file:

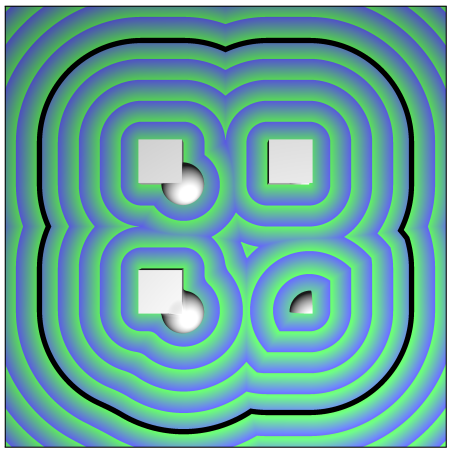
- `fragment_shader - Task1.glsl`



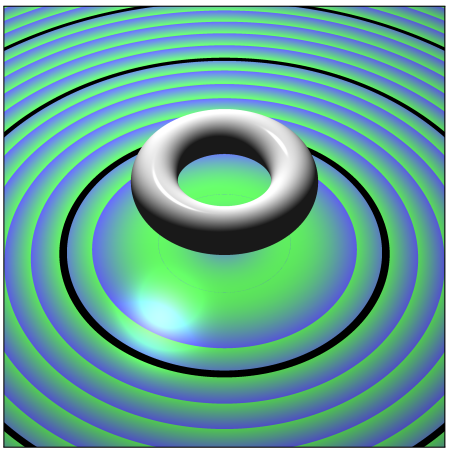
(a) Cube



(b) Geometric operations

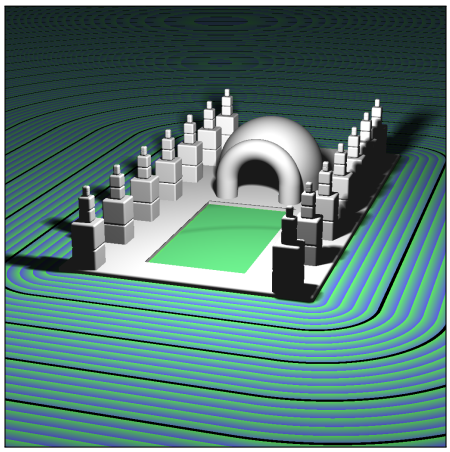


(c) Signed distance texture



(d) Torus and specular lighting

(e) Soft shadows



(f) A building showing repetition

Figure 4: Reference images

- `fragment_shader - Task2.glsl`
- `fragment_shader - Task3.glsl`
- ...

A Two different ways to model a cube

The choice between seemingly-identical distance functions can actually have significant impact on the performance of your ray marcher. Remember that

- your ray marcher takes the largest step it can based on how near the nearest object could be to the current point
- a valid SDF need only return an upper bound on the signed distance, not the minimum such upper bound

With that in mind, here are two equally viable ways to implement a cube as an SDF:

Non-linear form:

```
float cube(vec3 p) {
  return max(max(abs(p.x), abs(p.y)), abs(p.z)) - 1; // 1 = radius
}
```

Linear form:

```
float cube(vec3 p) {
  vec3 d = abs(p) - vec3(1); // 1 = radius
  return min(max(d.x, max(d.y, d.z)), 0.0) + length(max(d, 0.0));
}
```

Both methods return an upper bound of the distance from point `p` to the unit cube. Where they differ is in their treatment of points which due not fall directly in front of one of the six faces of the cube. In the non-linear form, the upper bound on the distance from the cube to the point is the maximum of the length of the projection of the point along the three axes. In the linear form, the upper bound is determined by the vector length of `d`. So when a point is out in space past the corner of the cube, you can think of the non-linear form as returning the biggest side of the triangles from the corner to the point; whereas the linear form returns the hypotenuse of those triangles. The linear form will thus return values which are larger than those of the non-linear form.

If we visualize the signed distance for these two functions side-by-side, one can see that the linear version's distance field is 'tighter' to the model than the non-linear form (Figure 5). This means that a ray marcher using the linear form

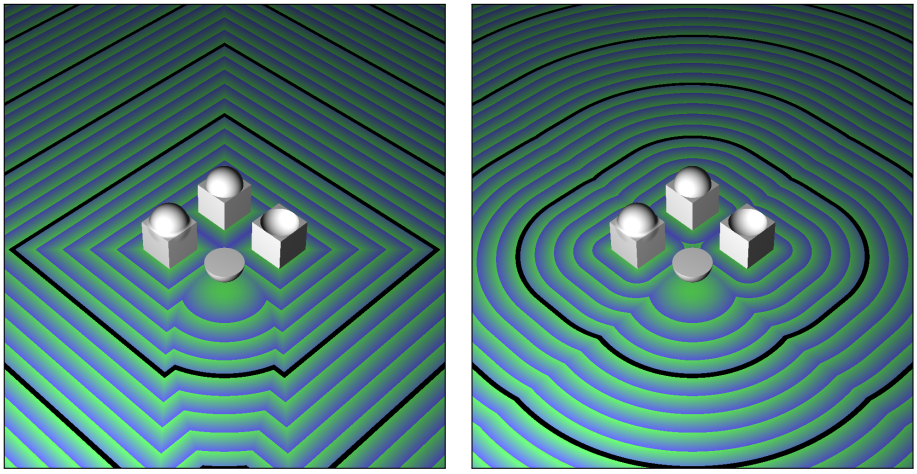


Figure 5: Side-by-side comparison of two implementations of the `cube()` distance function. On the left, by choosing the absolute maximum distance along each axis, we return unnecessarily far distances along the axes |X|=|Z|. On the right, by choosing length(p), we achieve a tighter upper bound on the distance function along the |X|=|Z| lines.

will be able to take larger steps from nearer in, and thus take fewer iterations overall to render the cube.

A second benefit to the linear form is that its gradient varies continuously. This is a nice trait when blending primitives together into more complex shapes; we avoid introducing regions where the gradient to the surface changes discontinuously, which could cause rendering artifacts.

Since the linear form has continuous gradient, what happens when we increase the radius of the function? Try increasing the radius by subtracting 0.5 from the final SDF (Figure 6).

B Twists and other nonlinear transformations

Let's say that you want to model a box of height 4, which twists along the Y axis and which tapers inwards (becomes skinnier) as it rises. The result would look a bit like a unicorn horn, if unicorn horns were square at the top instead of pointy (Figure 7).

B.1 Twist

A *twist* deformation is a nonlinear spatial transform which rotates geometry by a variable amount. Given an axis, geometry in the plane perpendicular to that axis is rotated as a function of how far along the axis the plane lies. Your goal

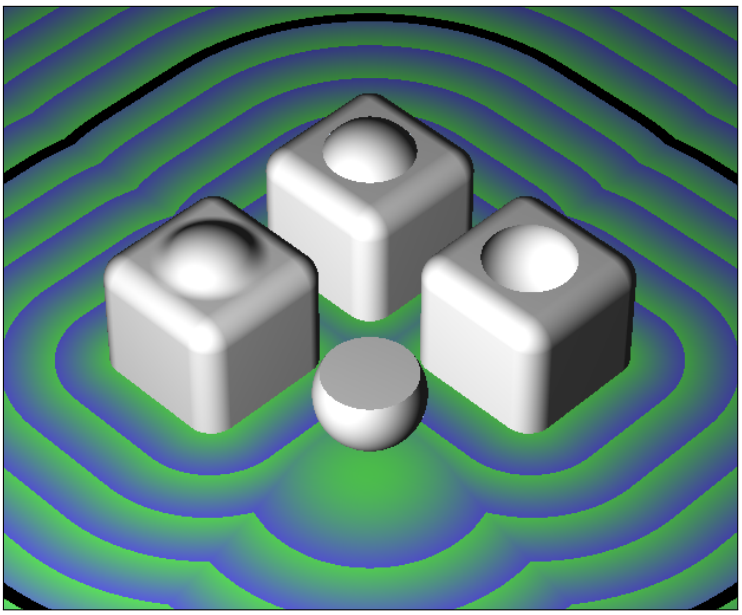


Figure 6: Using the linear version, `cube() - 0.5` creates a box with rounded edges and corners

is a twist along the Y axis, so you'll be rotating the X and Z coordinates of your points.

```
float twistedBox(vec3 pt) {
  float t = pt.y * PI;
  return box(vec3(
    pt.x * cos(t) + pt.z * sin(t),
    pt.y,
    -pt.x * sin(t) + pt.z * cos(t))) / (2 * sqrt(2));
}
```

which implements the classic rotation matrix,

$$R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

Remember that you're not actually twisting *geometry* with this rotation: instead, you're twisting the *space* that your geometry occupies. This can lead to complications in ray marching, because sometimes the estimate of 'nearest distance' to a twisted surface will underestimate the distortion of space. If you see glitches and visual artifacts in your rendering–missing pixels–try dividing

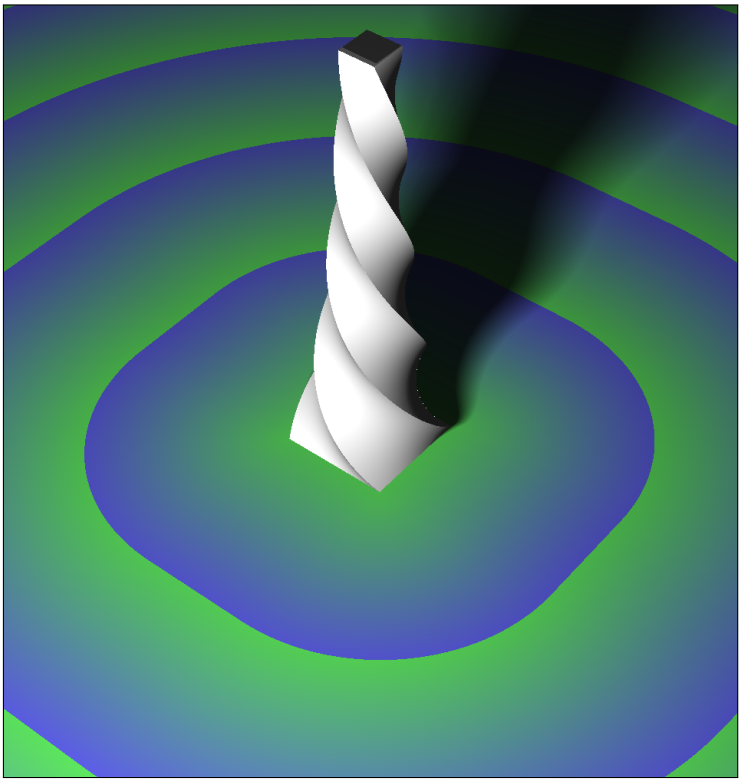


Figure 7: A twisted, tapering cube, which demonstrates the utility of the *Lipschitz constant*.

your distance function by the greatest distance that a point could travel during a rotation. In this case, since you're twisting a unit cube, that distance is $2\sqrt{2}$ (twice the distance from the corner of the cube to the axis of rotation). This is an example of a *Lipschitz constant*.

For more on the Lipschitz constant (and many other tips and tricks for signed distance field modeling), check out Hart's 1996 paper.

B.2 Taper

A *taper* deformation is a nonlinear spatial transform with reduces the radius of geometry along an axis. This is accomplished by *increasing* the values of the X and Z coordinates of the point as a function of increasing Y. For ease of reading, we've separated the taper from the Y scaling:

```
float taperedTwist(vec3 pt) {
  pt.x *= (pt.y + 2);
  pt.z *= (pt.y + 2);
  return twistedBox(pt) / 1.25;
}

float tallTaperedTwist(vec3 pt) {
  return taperedTwist(vec3(pt.x, (pt.y - 3) * 0.25, pt.z));
}
```

Like the twist, a taper can create situations where a ray near your geometry will be given an overestimate of the distance to the nearest geometry, causing the ray to skip the surface. Here again we divide by a small constant to ensure that the ray does not overshoot. 1.25 is sufficient here.

C Gratuitous torus abuse

Frustrated by the final task? Want to vent your anger on a defenseless torus? Try this:

```
float twistedTorus(vec3 pt, float R, float r) {
  float t = sin(currentTime) * ((R+r)*(R+r) - pt.x*pt.x) * PI/8;
  pt = vec3(
      pt.x,
      pt.y * cos(t) + pt.z * sin(t),
      -pt.y * sin(t) + pt.z * cos(t));

  return torus(pt, R, r) / 5;
}
```