CS148 Homework 5

Homework Due: Aug 4th **(Friday)** 12PM - 2PM **or** 5PM-7PM

0.1 Assignment Format

This assignment has **3** TODO s that are covered within the first few pages of the document. The rest of the handout consists of instructional tutorials. The TODOs are a mix of coding and Blender GUI exercises.

0.2 Collaboration Policy and Office Hours

All policies from here on are the same as they were for HW1. See the HW1 document for details.

Quiz 4

The fith and last quiz will be held in conjunction with the live grading on Friday, Aug 4th. See the FAQ post on Ed for more details on the live grading sessions. You will be randomly asked one of these questions:

- What concerns may come up if we were to use Explicit (Forward) Euler to evaluate the equations of motion when updating the state of a physical system? How do these issues compare to those of Implicit (Backward) Euler?
- What is the difference between the Lagrangian fluid simulation approach vs. the Eulerian fluid simulation approach? What is the advantage of combining both approaches?
- How are springs relevant to cloth simulation? Describe the three types of springs that we need to construct the simplest cloth model.
- How can we force a collision in our physical simulation to be inelastic vs. completely elastic? How can we simplify the problem of modeling a 3D collision into a 1D problem?
- Describe one technique for generating the inbetween frames of a scene in motion given a set of manually crafted keyframes. Given a bunch of frames of a scene in motion, how do we ray trace motion blur?

1 Assignment

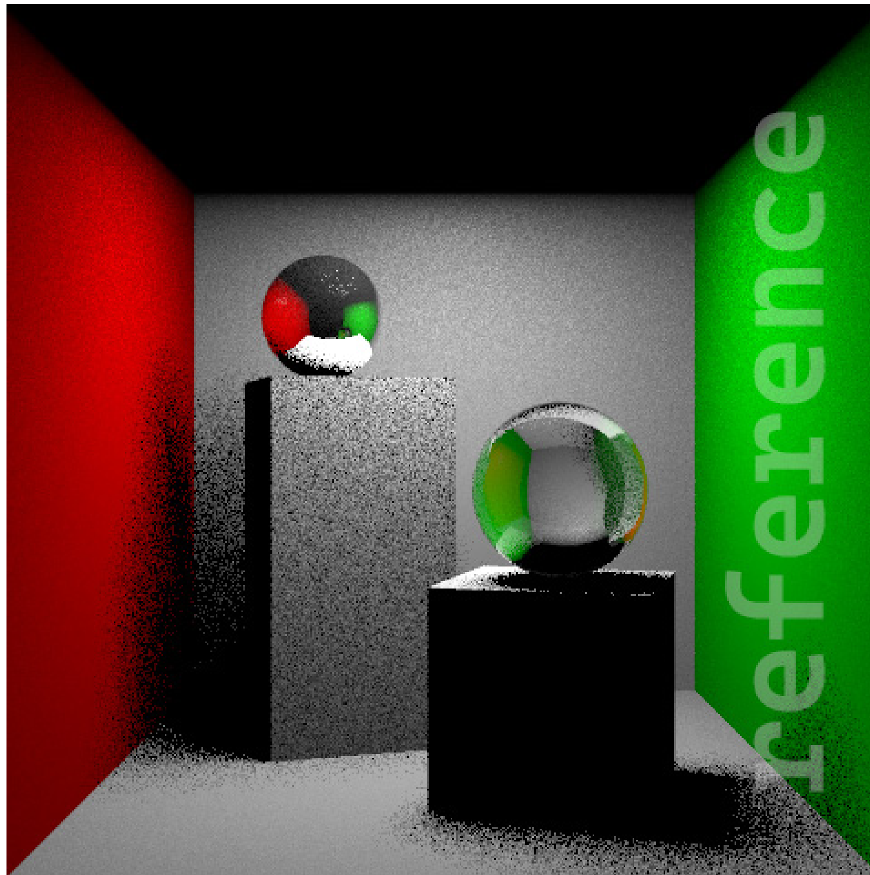
1.1 TODO 1 : Implement global illumination (for color bleeding).

Your task this TODO is to adapt the simple ray tracer from HW3 from using direct illumination to using global illumination for more natural lighting. To demonstrate the effect of global illumination, we will apply the simple ray tracer to the classic Cornell box scene.

We've already created the scene for you in Blender and have set up the code infrastructure to access information from the scene using Blender's Python API. **You can find the scene and code all** in this .blend file. Like in HW3, you will need to fill in a section of code that is further elaborated below. **Please start this part early! Depending on your computer hardware,**

the final image can take up to an hour to render! So plan ahead and give yourself enough time!

To run the code, you will need to run the `simpleRT_UIpanels` script first every time you launch Blender (like in HW3) before running `simpleRT_plugin`. You should try launching Blender from the command line as you did in HW3, since the starter code comes with a function that prints out the estimated wait time to the command line for how long the render will take to finish. Rendering the scene with the starter code should give you the following image:



Let's take a look at what the existing code already does:

1. Scroll through the code that makes up the `RT_trace_ray` function, and you'll see a complete implementation of HW3 with one key addition – from lines 63 to 76, there is an implementation of area lights that we did not have you do in HW3. Notice how the area light code uses randomness to compute or **sample** a point on the area light. As we discussed in lecture, we can model an area light as a collection of point lights, and when we ray trace, we pick random points on the area light to which we shoot our shadow rays.

2. To understand how the area lights are sampled in code, scroll to lines 174 to 202 – notice the triple for-loop starting with a loop over the number of samples. The naive way to implement sampling with ray tracing is to include the loop over every pixel of our image plane within another loop over every sample. For each sample, we ray trace the scene. As we ray trace, anything in the scene that requires sampling will involve some degree of randomness – in the case of area lights, we pick a random point on the area light to shoot our shadow rays.

Suppose we have s samples – then this procedure gives us s ray traced scenes, each with their own degree of randomness for the area light shadows. If we average all these ray traced scenes – which is what line 196 is doing with a running average (using a "buffer" variable to store the average) – then we will get the intended soft shadow effect that area lights naturally cause in real life.

3. Notice on the lower-right of the GUI we have the samples count set to 4. So our image is the result of averaging only 4 randomly sampled ray traced scenes, hence why our shadows look so noisy. If we were to increase the number of samples, then we will eventually get rid of the noise for more solid looking shadows. For the purposes of doing this HW though, you'll want the sample count relatively low so that Blender doesn't take forever while you're debugging.

4. You might notice parts of the code referring to "Corput", short for Van der Corput. We're not going to cover it in this class, but the Van der Corput sequence is basically a series of numbers that lead to a more efficient sampling technique, allowing our code to converge faster. In the case of area light sampling, converging faster means we take fewer samples to get more solid looking shadows.

   The idea is that we vary the point through which we shoot of our camera rays – we don't always simply shoot the camera rays from the camera position through the center of each pixel. Instead, notice how lines 184-186 offset our ray away from the center of the pixel by some Corput factors. It turns out that by doing these offsets, we will take fewer samples to get a more complete image. Feel free to look it up if you're interested! As a fun fact, Blender Cycles uses a similar technique, though it uses what's called the Sobol sequence instead of the Van der Corput sequence.

The rendered image so far has "dead" black shadows, i.e. the colors from the red and green walls do not "bleed" onto the cubes. This is because we only have direct diffuse and specular, i.e. the diffuse and specular rays stop once they hit an object. In reality, the object receives light from not only light sources, but also from other objects in the scene. You have heard of this phenomenon as color bleeding from lecture.

To mimic real-world lighting, we need to add more bounces for the rays to achieve global illumination. We will do so for the diffuse rays in this HW by giving them recursive bounces to implement the scattering procedure we discussed in lecture. We will ignore global illumination for the specular component to keep the HW within reasonable length.

Find the `TODO` in the provided code, then:

1. First, check if `depth > 0`. Similar to reflection and transmission rays, we only shoot recursive rays when the recursive depth is greater than 0. Only proceed with the next steps if this true.

2. For the purposes of computing the recursive ray direction, we need to first establish a local coordinate system with the normal vector at the intersection point as the z-axis. That is, we need to find a pair of x and y axes so that the z-axis becomes `hit_norm`.

   For the x-axis, we will make it a unit vector orthogonal to the normal. Start by initializing this vector to an initial guess of (0,0,1). Now, if (0,0,1) is too close to the normal, then change it to (0,1,0) instead. Two vectors are "too close" if they are almost parallel, i.e. the magnitude of their dot product is close to 1. (Self-check: do you know why? Ask in office hours if you're not sure!)

   We then compute the normal-direction component of x, which can be computed as $x \cdot n * n$ where x and n are the x and normal vectors respectively. Essentially, this is just a dot

product of the x and normal vectors, then a component-wise multiplication with the normal vector. Subtract the result from the x vector to make it orthogonal (aka perpendicular) to the normal, then normalize x. Some of you may recognize this process as the Gram Schmit orthogonalization technique from linear algebra.

The y vector can be obtained via the cross product of the x vector and the normal. Python has a `.cross()` function.

3. Also for the purposes of computing the recursive ray direction, we need to sample a hemisphere as discussed in lecture when we introduced the idea of scattering rays. We first orientate this hemisphere at (0, 0, 1). Imagine the top half of a sphere centered at the origin of the xyz-axis, where up is positive z. We want to randomly pick a vector direction along this hemisphere.

   Mathematically, we can express a point on the hemisphere as $[\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta)]$, with $\theta \in [0, \pi/2), \phi \in [0, 2\pi)$. We uniformly sample on the hemisphere surface by making $\cos(\theta)$ a uniform variable between 0 and 1.

   Computationally, we do this by first creating two random variables r_1 and r_2 between 0 and 1, and let r_1 be $\cos(\theta)$ and $r_2 * 2 * \pi$ be ϕ. From here, you can simply plug these values appropriately into the above formula for a point on the hemisphere (remember that $\sin^2(\theta) + \cos^2(\theta) = 1$). Since the hemisphere is centered at (0,0,0), this point is also a ray direction. (Self-Check: do you know why? Ask in office hours if you're not sure!)

4. The ray direction that you computed from the previous step is within some abstract local space of a hemisphere, and thus needs to be transformed into world space to actually be used. To do this, we use the coordinate system computed in Step 2 to determine a matrix transform. This transform will take the ray from this abstract hemisphere and place it alongside the normal of our object in world space.

   Let `x, y, n` be the coordinate system you computed in Step 2. Then to form the necessary matrix transform in Python, you can do:

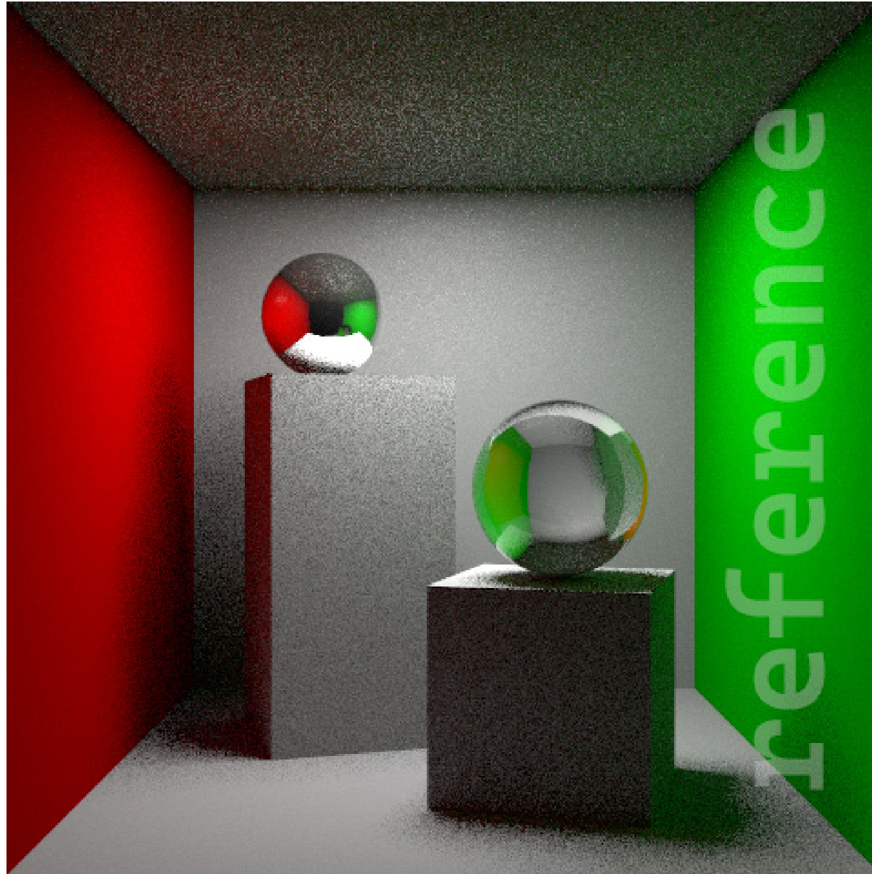
   ```
   mat_transform = Matrix()
   mat_transform[0][0:3] = x
   mat_transform[1][0:3] = y
   mat_transform[2][0:3] = n
   mat_transform.transpose()
   ```

   Use this matrix to transform your ray result from Step 3.

5. Finally, recursively trace the ray from Step 4 as you did for the reflection and transmission rays from HW3, remembering to take into account self-occlusion. Store the result as the indirect diffuse color. This color needs to be scaled by r_1 to account for face orientation as well as `diffuse_color` to account for absorption. Add the final result to `color`.

After finishing the above steps, your render should look similar to the following. When you think you have your code correct, **render your final image at 480x480 100% resolution with 16 samples and a depth of 3. This may take up to an hour or more. Please save this render for grading.**

Show us: (2 pt) The render as described above.



In case you are curious, this is how the scene from HW3 looks if rendered with the completed global illumination code. Notice the softer shadows and color bleeding. You do not need to generate this image yourself.

1.2 TODO 2: Render a scene with geometry from a fluid or cloth simulation.

In lecture, we talked about both fluid simulation and cloth simulation. In this PDF, you'll find tutorials below on how to set up simple examples of both in Blender. For the purposes of grading, pick either fluid or cloth simulation (or both if you really want!) and use it to generate geometry that you then place into a scene with appropriate lighting and materials (ideally the scene(s) that you generated last HW and are building for your final project).

> **Show us: (2 pt) A ray traced image using Blender Cycles of your own scene that has geometry generated from a fluid or cloth simulation (or both!) with appropriate looking lighting and materials.**

1.3 TODO 3: Render a scene using an advanced feature of Blender Cycles.

In lecture, we talked about how we would implement depth of field and motion blur, and later in the class, we'll go over how to implement volume rendering. Each of these phenomenon can be done in the Blender GUI with a few simple button clicks. You can find the respective tutorials further below in this PDF. Pick one of these (or multiple if you want!) to add to a scene that has a non-trivial arrangement of geometry, lighting, and materials.

NOTE: While we haven't discussed the theory of how we model and implement volume rendering and will not until after this HW is due, you don't actually need to know those details to enable volume rendering in Blender. For now, if you are interested in using volume rendering for your final project, then take the time to experiment with the Blender GUI options. We will talk about what's going on under the hood later.

> **Show us: (1 pt) A ray traced image using Blender Cycles of your own scene (with non-trivial geometry, lighting, and materials) that has one (or multiple) of depth of field, motion blur, and volume rendering enabled.**

2 Creating Geometry via Fluid Simulation

Blender provides a relatively easy-to-use interface for generating workable meshes from fluid simulation(s). Let's try it out with a simple example. First, taking the default Blender scene:

- Scale the default cube to enlarge it. This large cube will become our fluid domain.
- Add a smaller cube mesh inside the default cube. This small cube will act as our collision object.
- Add an UV sphere mesh above the smaller cube and inside the outer cube. This sphere will contain our liquid that gets simulated in our fluid domain.

It may help to switch to wireframe view in the Viewport toolbar when setting this up. See Figure 1 for a visual of the full setup.

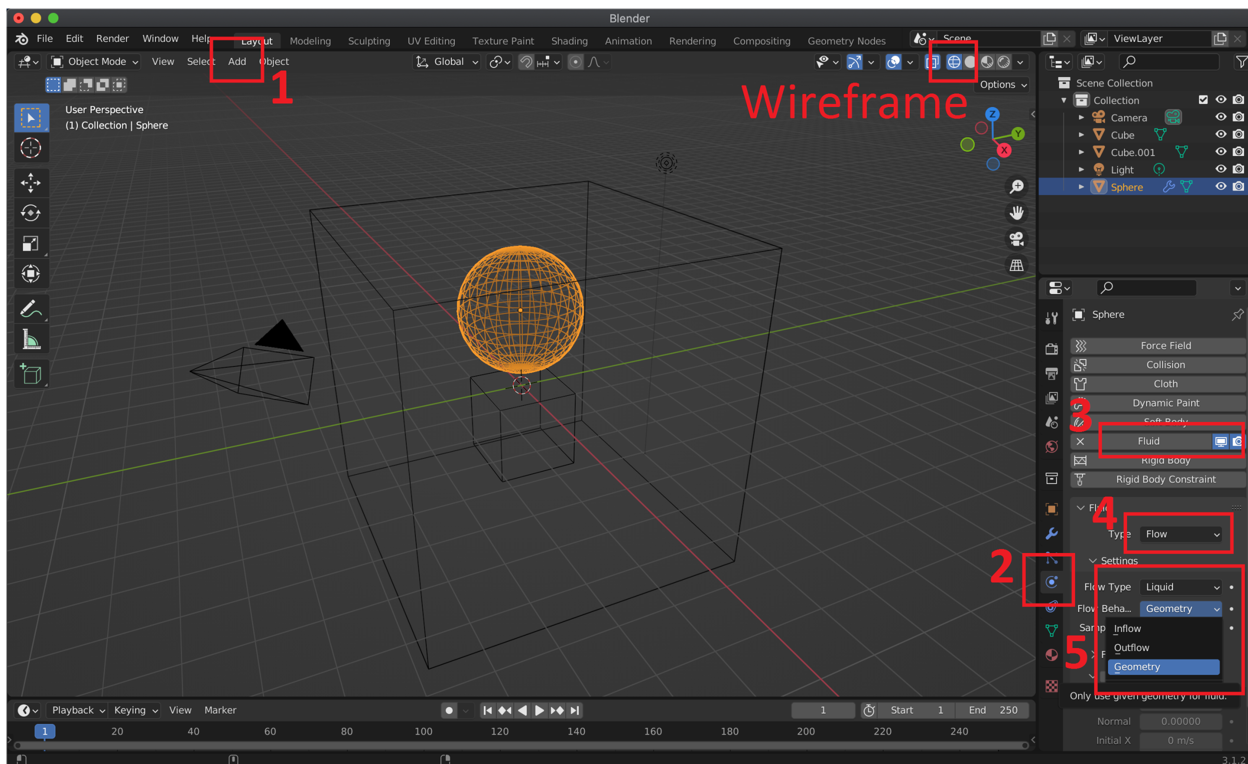


Figure 1

Start by selecting the sphere, then:

- Click on the **Physics Properties** panel of the Properties Editor (#2 in Figure 1) to access the physics options.
- Click on the **Fluid** option (#3 in Figure 1).
- Click on the **Type** dropdown menu and select **Flow** (#4 in Figure 1).
- Click on the **Flow Type** dropdown menu and select **Liquid** (#5 in Figure 1).
- Click on the **Flow Behavior** dropdown menu and select **Geometry** (#5 in Figure 1).

This sets up the sphere to be a giant collection of liquid particles. The **Geometry** option we set at the end tells Blender to have these liquid particles act as geometry when interacting with other objects in the scene (i.e. they will behave like actual mass when colliding with objects). In comparison, the **Inflow** and **Outflow** options will continuously add or delete fluid particles during the simulation, with the inflow simulating a running faucet, and the outflow simulating a drain with fluid disappearing upon interacting with the world.

Now, let's add a collision object to make the simulation more interesting. Select the smaller cube, then:

- Click on the **Physics Properties** panel of the Properties Editor.
- Click on the **Fluid** option.

- Click on the Type dropdown menu and select Effector (#1 in Figure 2).
- Click on the Effector Type dropdown menu and select Collision (#2 in Figure 2).

This sets up the cube to be an object that the sphere of fluid will collide with as it falls down in our simulation.

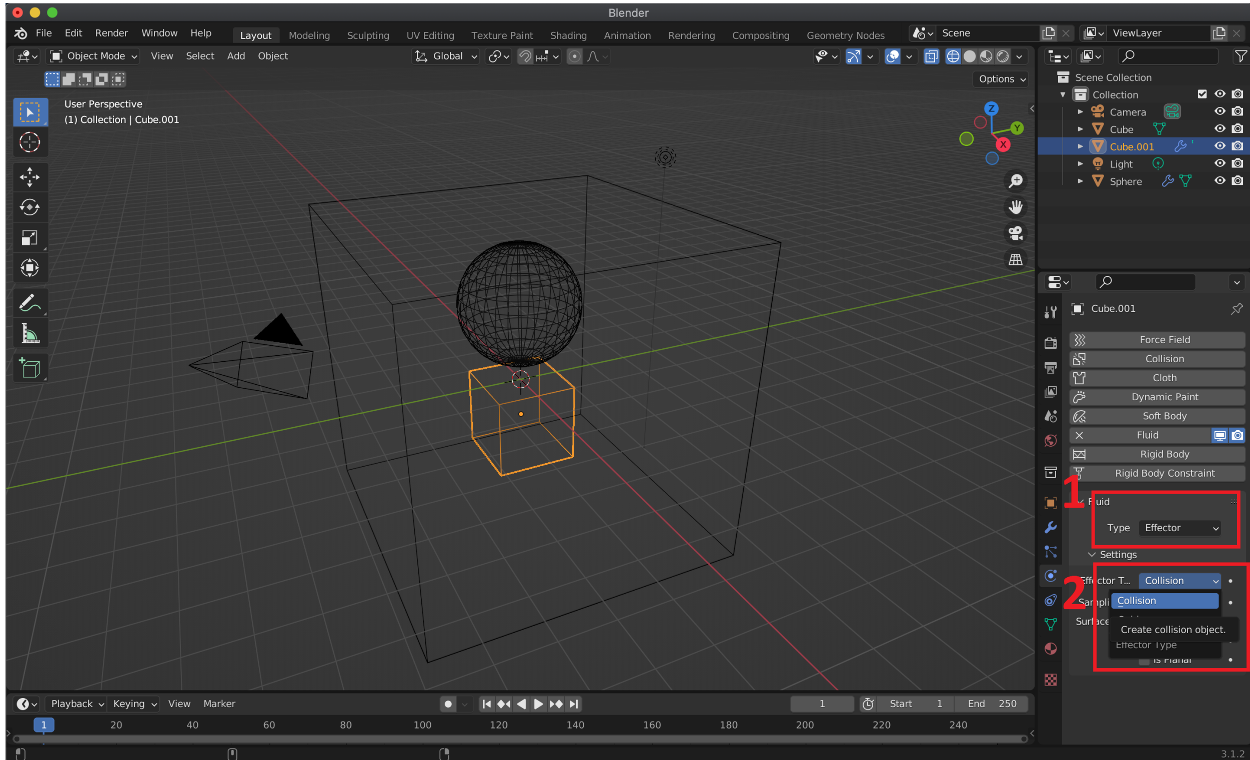


Figure 2

Finally, we need to set up the last cube in the scene. Select the large cube, then:

- Click on the Physics Properties panel of the Properties Editor.
- Click on the Fluid option.
- Click on the Type dropdown menu and select Domain to make this cube our fluid domain (#1 in Figure 3).
- Click on the Domain Type dropdown menu and select Liquid to tell Blender that this fluid domain will be used for simulating liquids (#2 in Figure 3).

This sets up the large cube to have a gravity field that will act on any liquid objects within. This includes the sphere of liquid, but not the collision cube. This means that when we run the simulation, gravity in our fluid domain will cause motion for the sphere of liquid and make it fall along the negative z-axis. But, the collision cube will stay in place. However, the sphere of liquid and the collision cube will interact, because we set the cube to be a collision object.

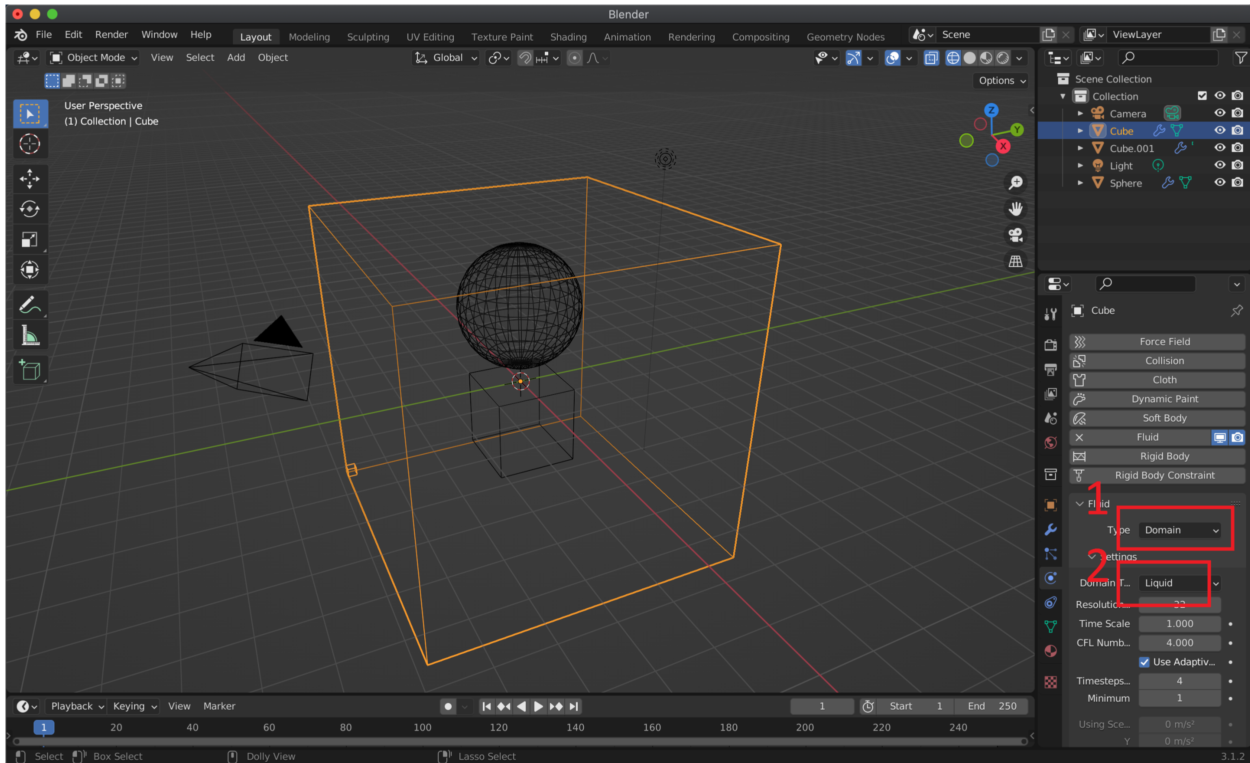


Figure 3

To actually run our fluid simulation, we need to set a few more parameters:

- Scroll down further in the **Physics Properties** panel of the large cube that we set to be our fluid domain.
- Set the **Cache** folder (#1 in Figure 4) to a folder where you want Blender to cache or temporarily save the files needed to model this fluid simulation.
- Set the **Type** for the simulation to **All** to tell Blender to generate geometry for all frames of the simulation (#2 in Figure 4).
- Check the **Mesh** checkbox to tell Blender to explicitly generate mesh geometry for the fluid simulation (#3 in Figure 4).
- Set the final **End Frame** of your fluid simulation in both the Properties editor and the **Timeline Editor** at the bottom of the Blender interface (#4 in Figure 4).
- Click **Bake All** when you're ready to generate your fluid simulation (#5 in Figure 4). It may take a few seconds for Blender to finish computing. There should be a progress bar at the bottom.

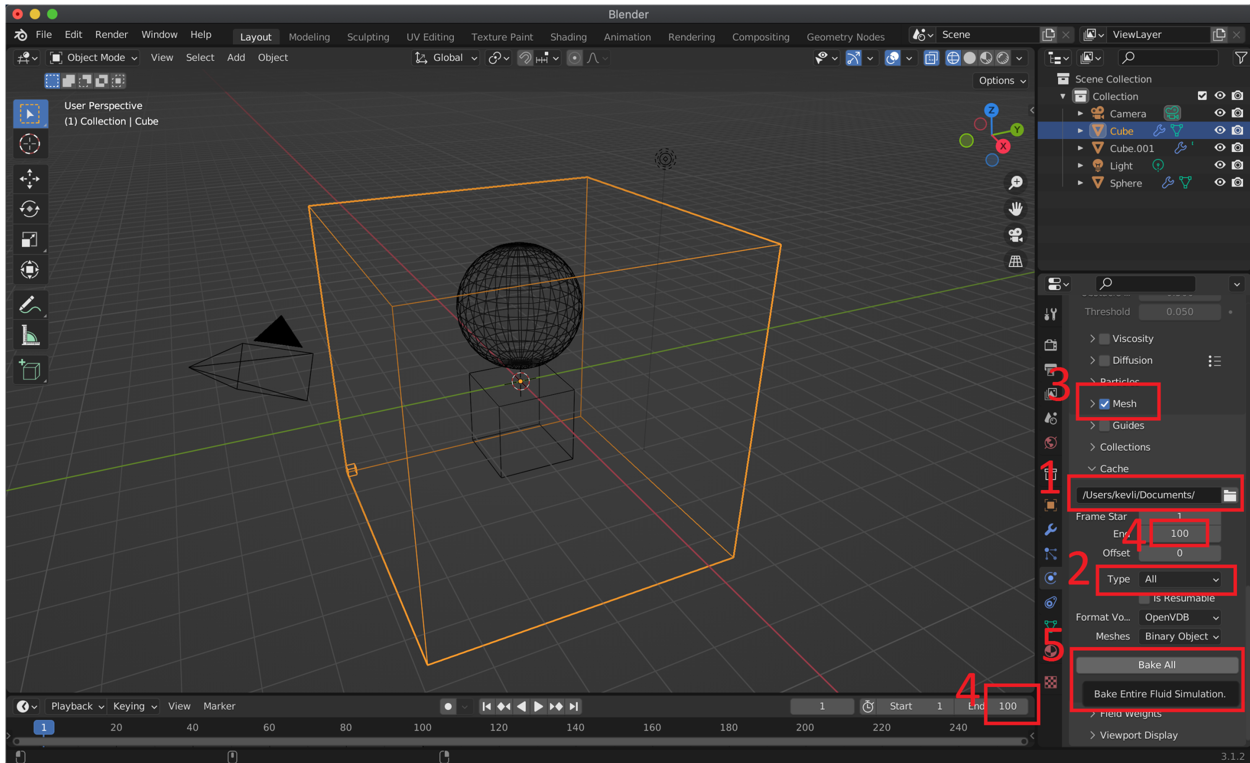


Figure 4

To see your fluid simulation in action, move the **Timeline Editor** at the bottom of the Blender interface up until you see the blue line clearly (see Figure 5). You'll see a timeline starting at 1 and ending at the **End Frame** you set earlier. You can scrub through the timeline by clicking and dragging the blue line to see the fluid simulation at different frames. For instance, Figure 5 shows us looking at the 50th frame of our simulation of 100 frames.

You may want to switch to solid view in the Viewport toolbar to get a better sense of the geometry of your fluid. If you want to redo your fluid simulation with a different set up, then you need to first delete the cache files for the current simulation (see Figure 5) before modifying your scene.

When you find a frame in your simulation that you're happy with, you can export all the geometry in that frame as an .obj mesh. Simply use the **File → Export** option. Then, you can import that same .obj into your actual scene and work with it like any other geometry. For instance, in Figure 6, we show how we can transform the generated liquid mesh just like any other object. You can also add materials, textures, etc to your liquid mesh.

For more info on the parameters and the various options that Blender provides for fluid simulation, see their official manual. This Youtube tutorial on using the "Quick Fluid" option in Blender may also be useful, especially for finding an appropriate material to give your water (in this case, the glass BSDF).

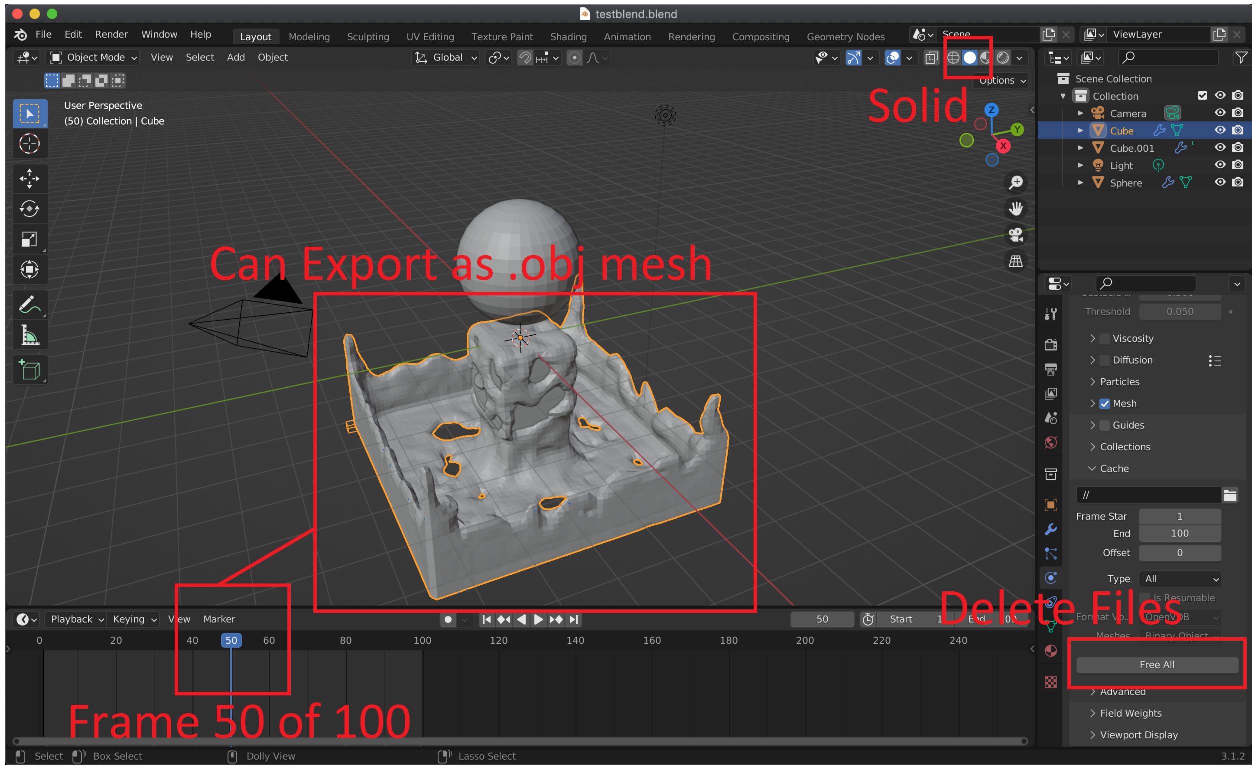


Figure 5

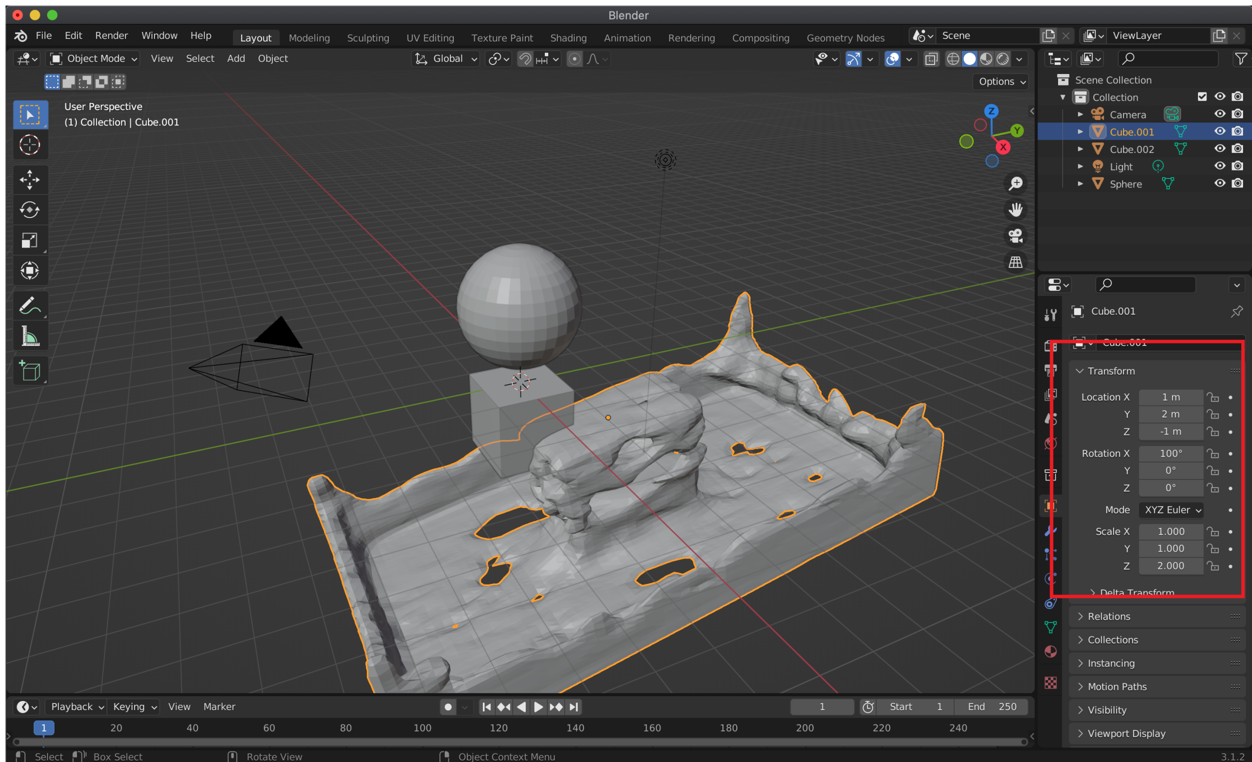


Figure 6

3 Creating Geometry via Cloth Simulation

Blender also provides a straightforward interface for deforming meshes with cloth simulation(s). Let's see with a simple example. Create a new scene, delete the default cube, and add a plane mesh plus the built-in **Monkey** mesh. Position the plane above the monkey head (see Figure 7) The plane mesh will become our cloth in our cloth simulation, and we'll have it collide with the monkey head.

Select the plane, then:

- Go into **Edit Mode**, then **Modifier Properties** in the Properties Editor (#1 in Figure 7).
- Click **Add Modifier** and add a **Subdivision Surface** modifier to allow us to subdivide the plane into a finer grid.
- In the modifier options, click **Simple** to use Blender's simple subdivision algorithm, which doesn't round out the edges of our plane (#2 in Figure 7).
- Right click the plane and click **Subdivide**. This will bring up a **Subdivide** control panel in the bottom left (#3 in Figure 7).
- Change the **Number of Cuts** to 50 to make the plane a 50x50 grid of squares.

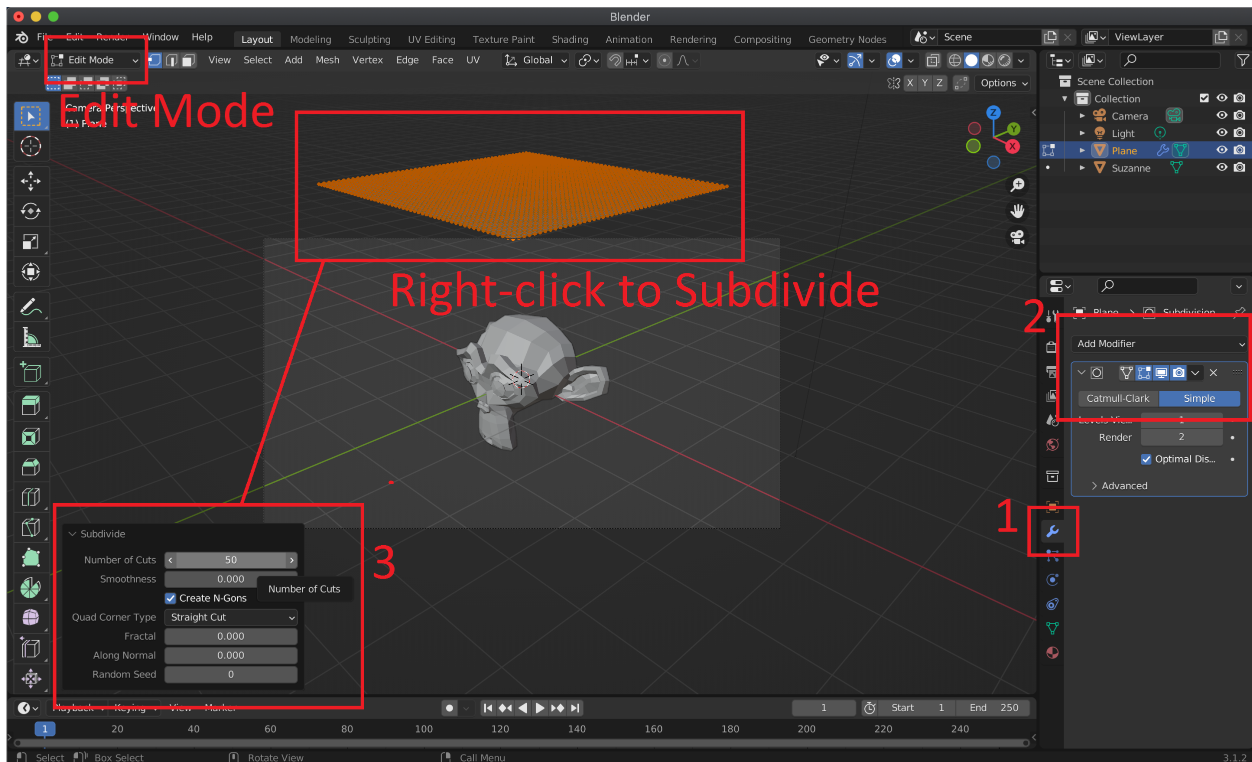


Figure 7

Now, we need to set the plane to act as a cloth object and the monkey head to act as a collision object.

- Select the plane (you want to go back to **Object Mode**), then click on the **Physics Properties** panel of the Properties Editor. Select the **Cloth** option. This tells Blender to treat our plane as a network of particles and springs to model the physics of a piece of cloth.
- Select the monkey head, then click on the **Physics Properties** panel of the Properties Editor. Select the **Collision** option. This tells Blender to have our monkey head interact with any objects that collide with it.

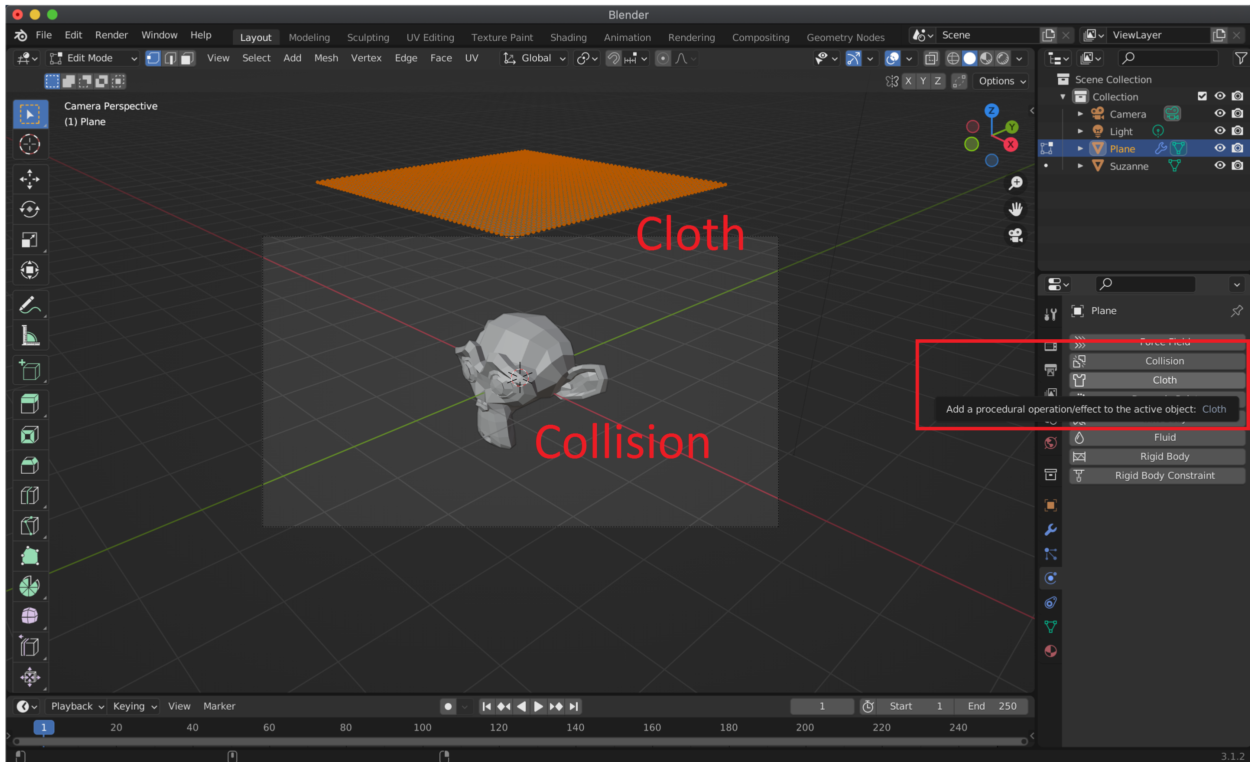


Figure 8

To see your cloth simultion in action, move the **Timeline Editor** at the bottom of the Blender interface up until you see the blue line clearly (see Figure 9). You'll see a timeline starting at 1 and ending at 250 (you can change this to another number like you did with the fluid simulation). You can scrub through the timeline by clicking and dragging the blue line to see the cloth simulation at different frames. For instance, Figure 9 shows us looking at the 31st frame of our cloth simulation.

To get a better sense of how your deformed plane looks like as a cloth, you can turn on **Shade Smooth** when right-clicking it in **Object Mode**. Make sure to change the **Render Engine** to **Cycles** for ray tracing. Then click the cycles render preview on the viewport toolbar (see Figure 9).

When you find a frame in your simulation that you're happy with, you can export all the geometry in that frame as an .obj mesh. Simply use the **File → Export** option as usual. Then, you can import that same .obj into your scene and work with it like any other geometry.

For more info on the parameters and the various options that Blender provides for cloth simulation, see their official manual. You may also find this quick curtain tutorial useful in combination with this curtain plus wind simulation tutorial.

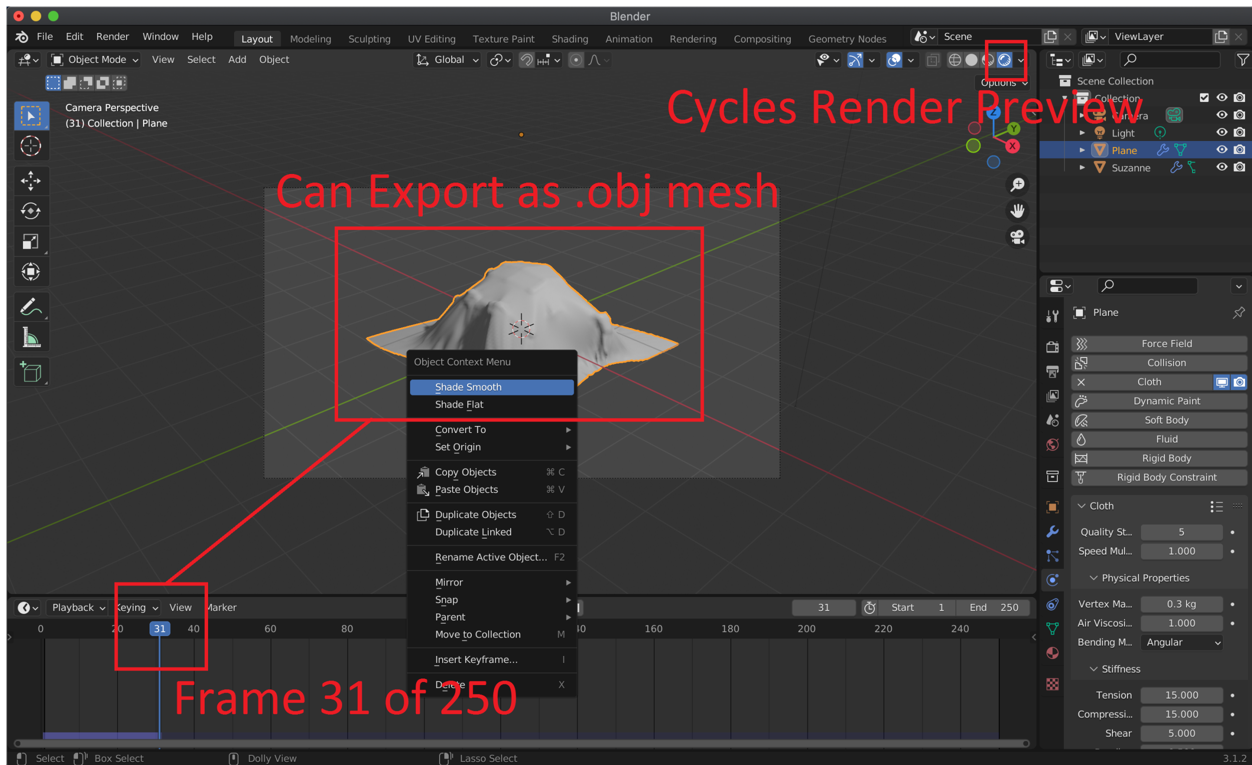
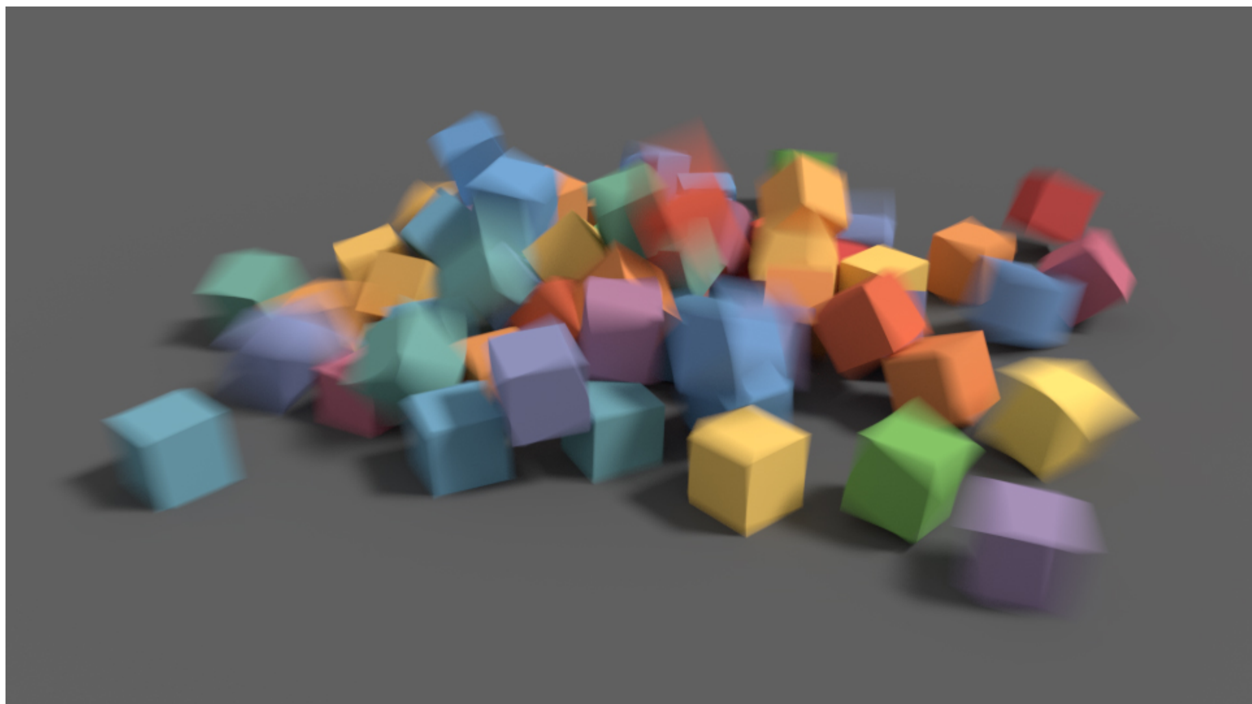


Figure 9

4 Motion Blur in Blender



Blender provides a very simple way to set up and ray trace motion blur. Let's take a look with a simple example. First, taking the default Blender scene:

- Translate the cube to some location that you want to act as its starting point.
- Move the **Timeline Editor** at the bottom of the Blender interface up until you see the blue line clearly (see Figure 10).
- Right click the cube and click **Insert Keyframe...** (see Figure 10), then click **Location** (see Figure 11). This will mark the current scene as our first position in the timeline and also tell Blender that we would like to interpolate location aka position values for an animation.
- Move the blue line in the **Timeline Editor** to a later frame like frame 10.
- Go back and translate the cube to another location that you want to act its next position.
- Then, using the same steps as above, insert a new location keyframe.

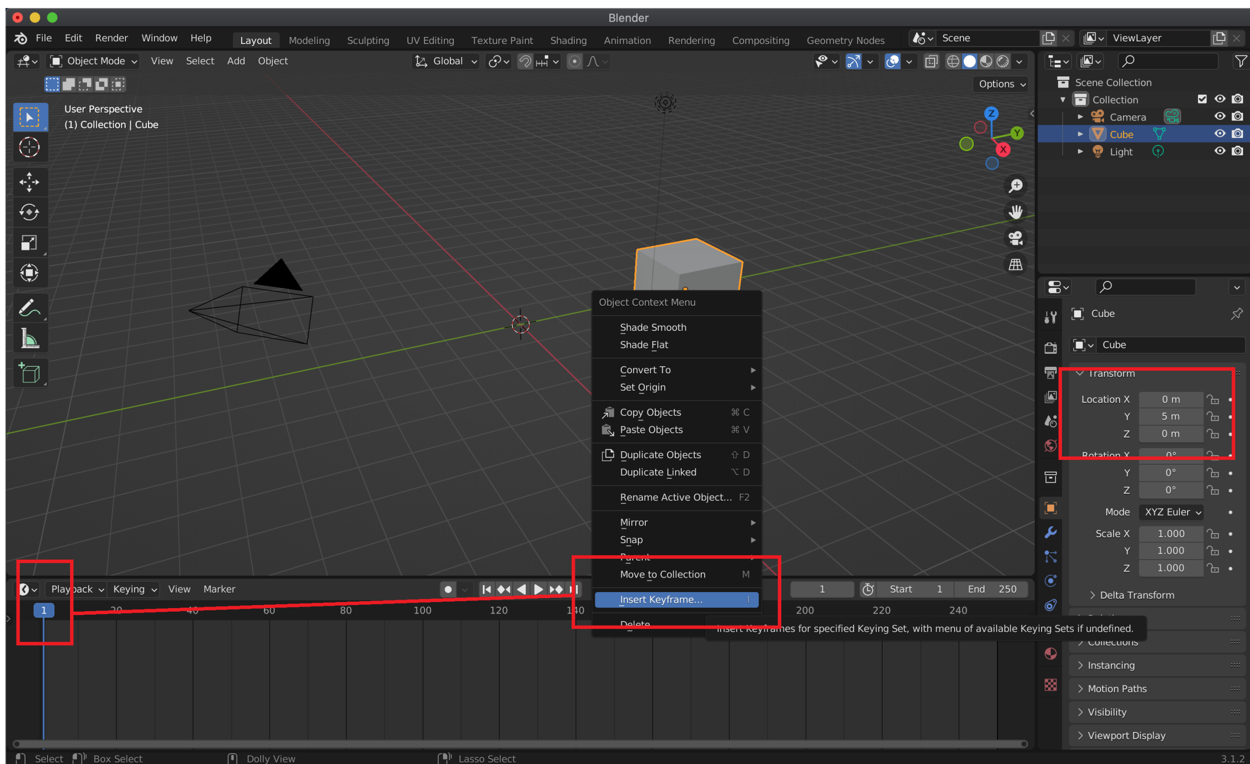


Figure 10

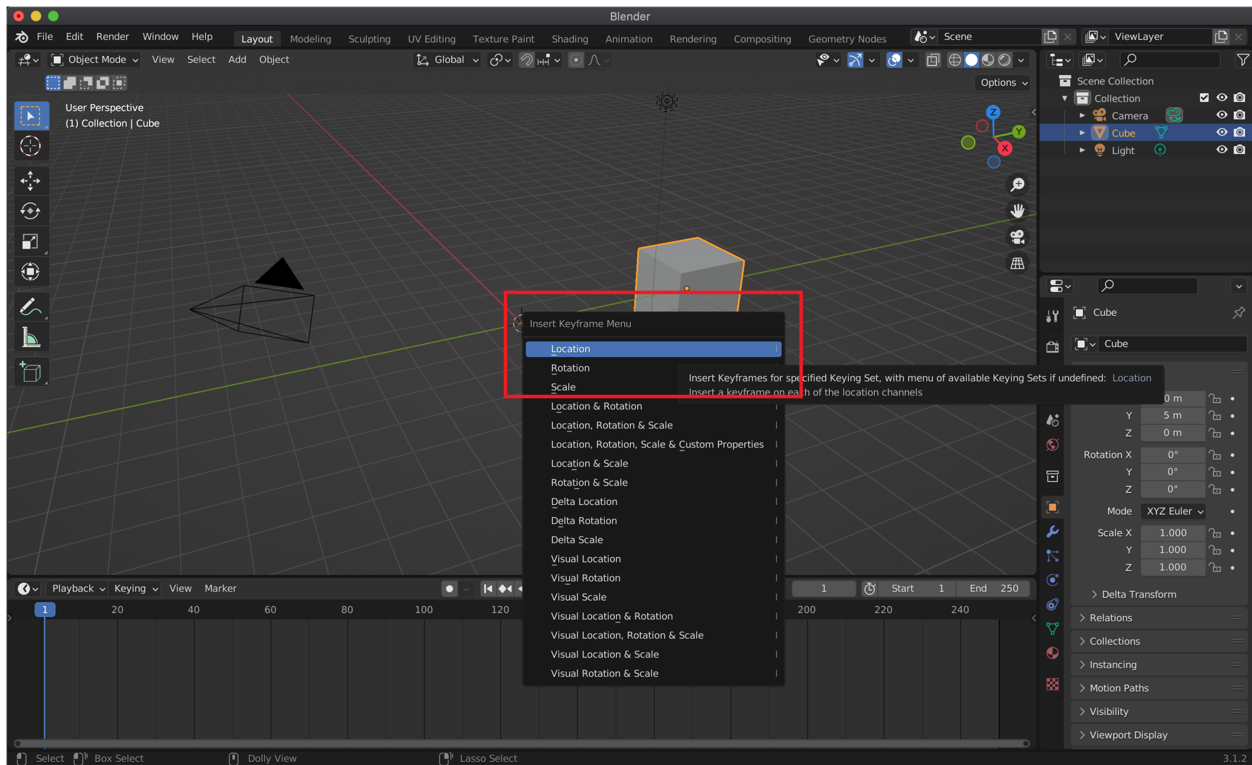
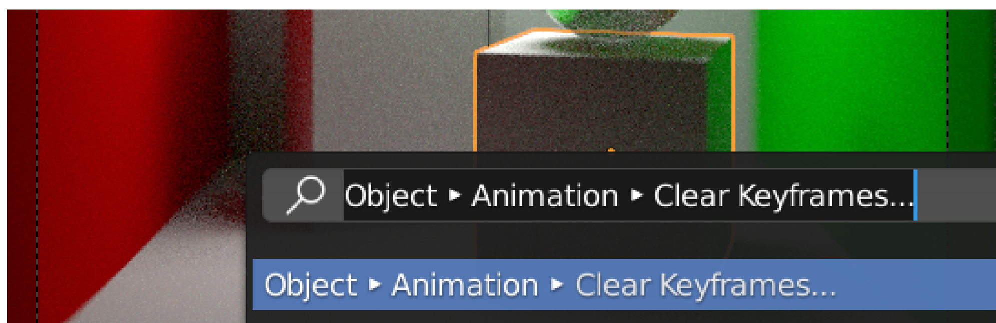


Figure 11

Now, if you scrub through the timeline by clicking and dragging the blue line from frame 1 to frame 10, you'll see your cube move smoothly from its initial position to its next position. Figure 12 shows our example with the cube at frame 7 of the animation and its interpolated position values.

To have Blender ray trace your scene with motion blur, first go to Render Properties as usual to change the Render Engine to Cycles for ray tracing. Then scroll down in this Properties Editor tab to find the Motion Blur checkbox (see Figure 13). Just check the box, and Blender will do motion blur! To see, pick a frame in the middle of the animation (e.g. frame 7 of 10 in the timeline) and render your scene as an image. You should see something like that in Figure 13 pop up in your Blender render window.

If you ever want to remove all the animation data that you created for the motion blur, then you can do so by selecting all the keyframes in the Timeline Editor and deleting them (right-click → Delete Keyframes). Alternatively, you can select your motion blurred object (e.g. the cube) and press F3 to pull up the Menu Search , then type Clear Keyframes .



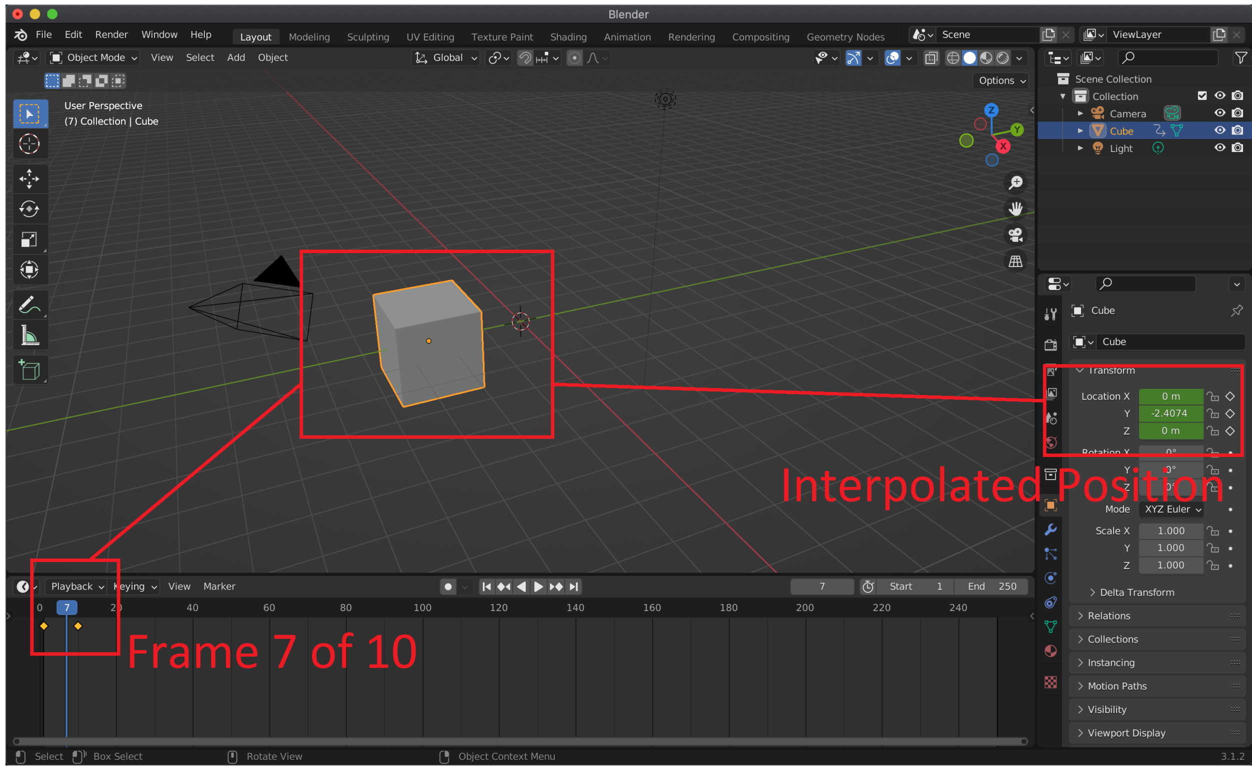


Figure 12

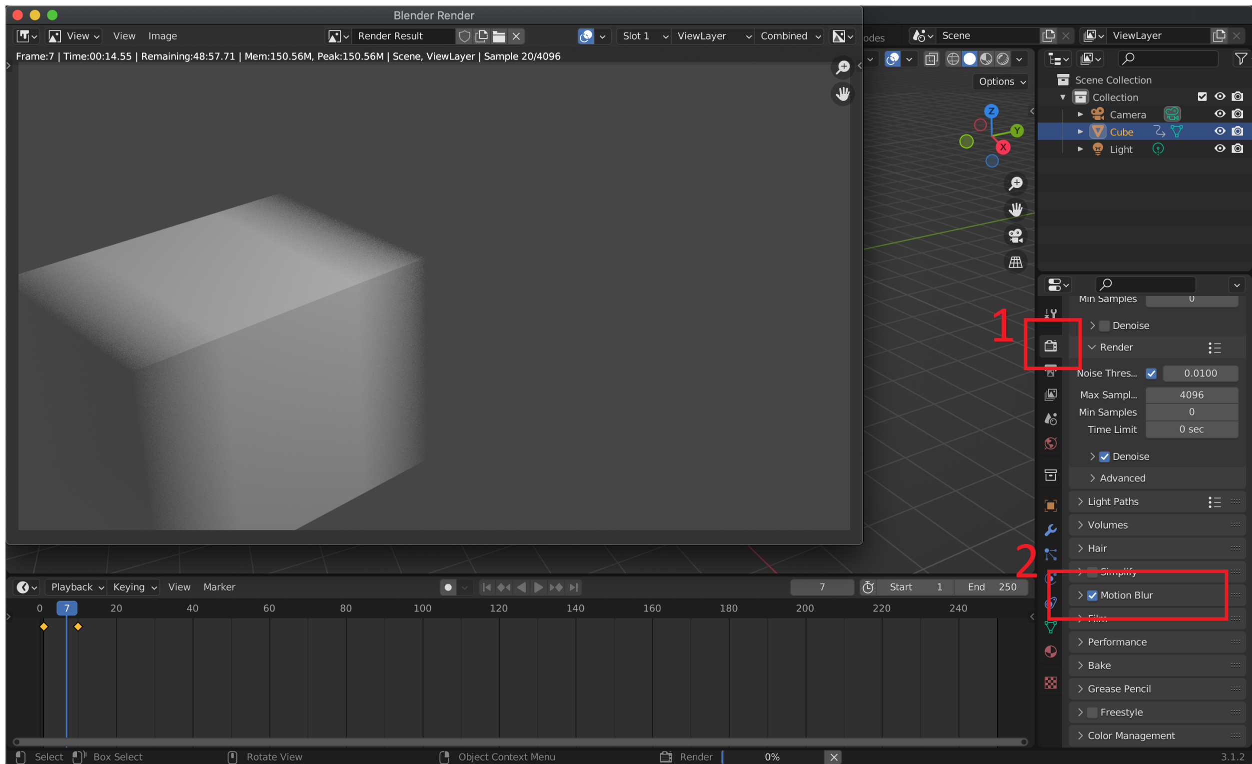
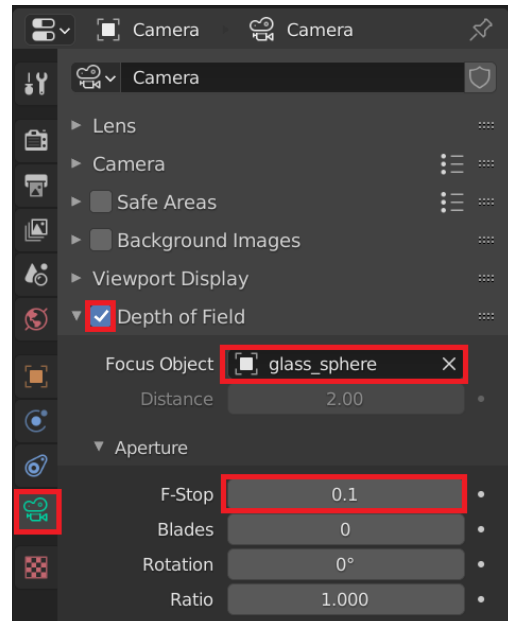


Figure 13

5 Depth of Field in Blender

Blender provides a very simple way to set up and ray trace depth of field. Let's take a look with a simple example. Open this .blend file containing the Cornell Box scene. Select the **Camera** in the **Scene Collection** in the upper-right of the interface. Then go to the camera's **Object Data Properties** in the **Properties Editor**. Click the checkbox for **Depth of Field**, and set the **Focus Object** to the **glass_sphere**. This will make our depth of field focus on the right-most sphere.



Try rendering the scene (or viewing it in the render preview) with and without this edit to see the effect of depth of field. Also try playing around with different values of **F-Stop** to get a feel for how it affects the strength of the blur. Lower values will increase blur, while higher values make the image more clear overall.

6 Volume Rendering in Blender

Volume rendering is an advanced technique for achieving effects that cannot be represented by surface meshes alone. More specifically, volume rendering is designed to render volumetric objects like smoke, fire, fog, and clouds. We will try out Blender's volume rendering capabilities with a simple fog example. For more info on all the options that Blender has to offer for volume rendering, see its manual on volumetric effects (It's for an older version of Blender, but should still be relevant).

To set up our fog example, create a new default scene and enlarge our default cube. Then:

- Go to **Material Properties** in the Properties Editor (#1 in Figure 14).
- Click on the **Surface** option (#2 in Figure 14), then **Remove** or **Disconnect** (#3 in Figure 14) the default Principled BSDF material.
- Scroll down to the **Volume** options (#1 in Figure 15).
- Add a **Principled Volume** in the **Volume** option.
- Then, set the **Density** (of our fog) to 0.2 (#2 in Figure 15)
- Add another cube inside our enlarged cube. You may want to switch to wireframe view when doing this.

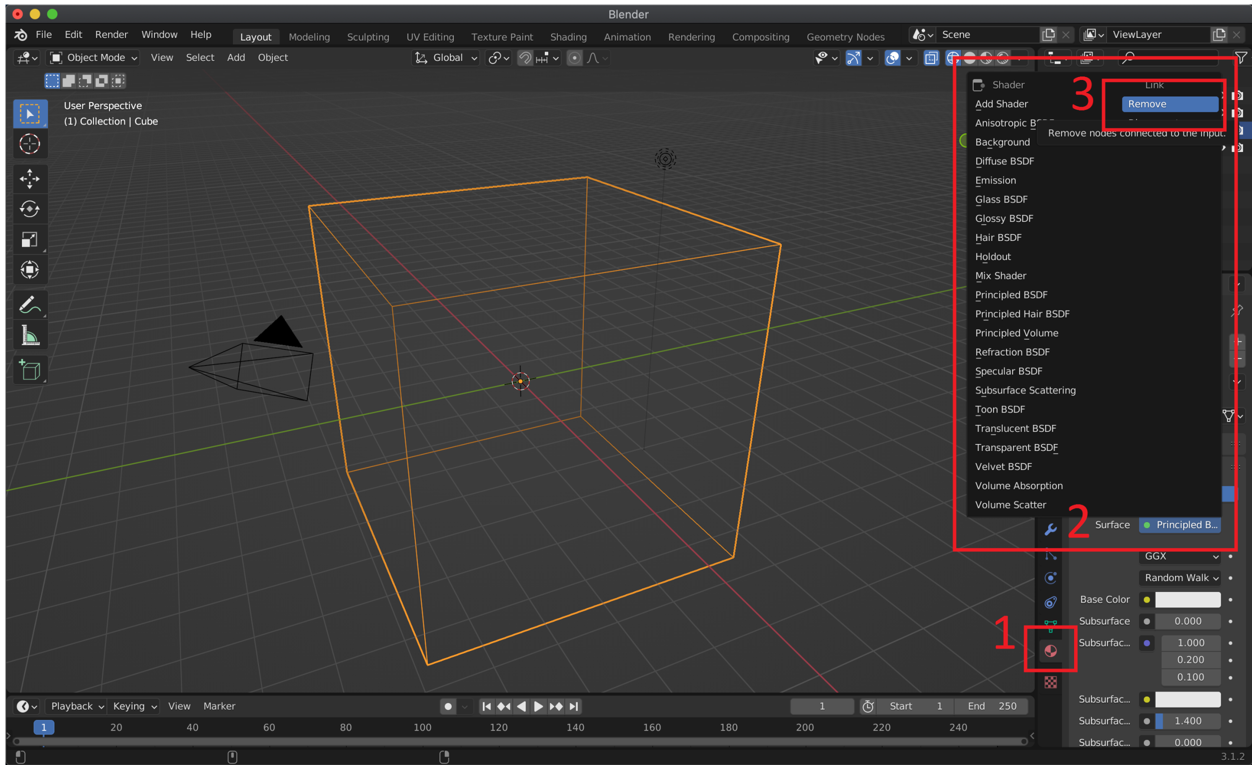


Figure 14

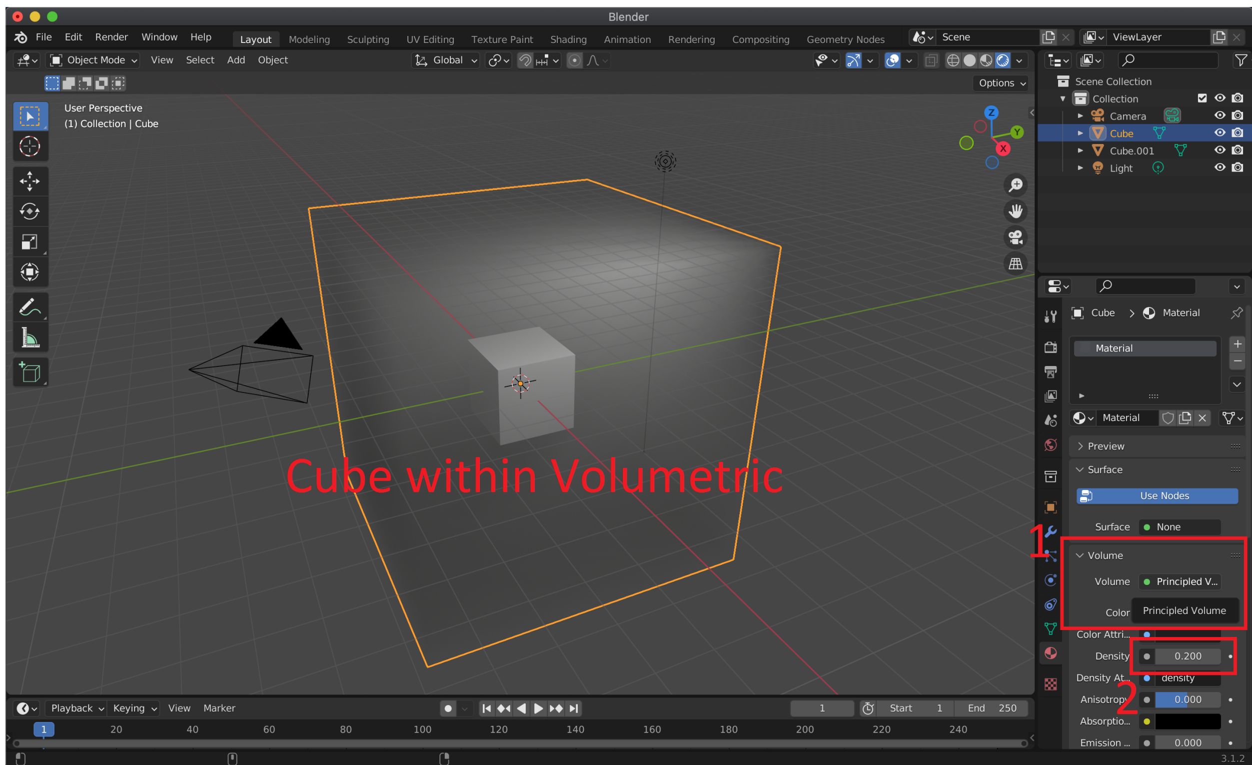


Figure 15

Set Blender to **Cycles** and toggle on the render preview. You should see a fog like effect inside the outer cube surrounding your inner cube. Render the image to see something similar to Figure 16.

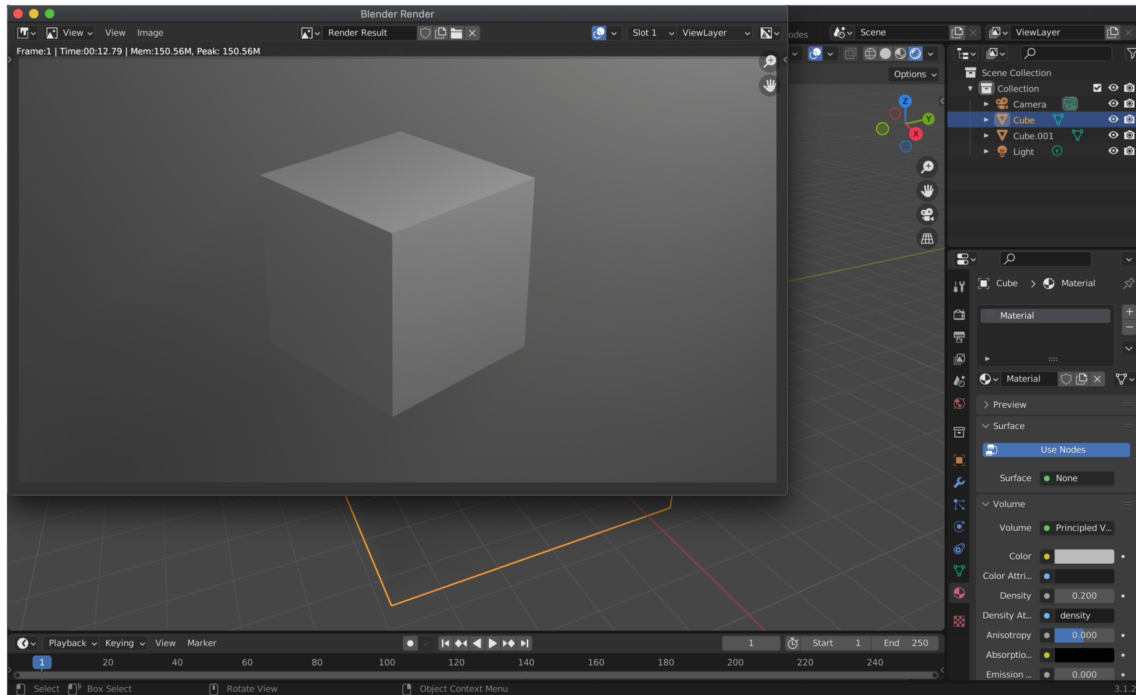


Figure 16

Try adding a fog effect to the Cornell Box scene from the previous section – it results in a pretty interesting visual phenomena. First, we need to surround the entire Cornell Box scene with a volumetric cube. You can do this by adding a new cube, translating it to $(-2.75, 2.75, 2.75)$, and scaling all its dimensions by 4. When adding the material, you will have to create a **New** material first. When you render the Cornell Box scene with fog, you might notice that the shape of the light becomes visible for reasons we will discuss later in lecture!