Introducing Assignment 5: Cloth Simulation

COS 426: Computer Graphics (Spring 2022)

Yuting Yang, Caio Costa

Agenda

- Administrative Notes
- Overview of A5
  - GUI
  - Tips
- Cloth Simulation
  - Constraints, Forces, and Intersections
  - Event Handlers
  - Optional Extensions

What’s Next?

- A4 due Sun, Apr 10 at 11:55pm
- A5 due Sun, Apr 17 at 11:55pm
  - Most of today’s focus
  - Should be released now
- Final Project
  - Proposals in-class Apr 21
  - Submission May 3
  - Presentations May 5 (TBD, likely 2 sessions)

What's Next?

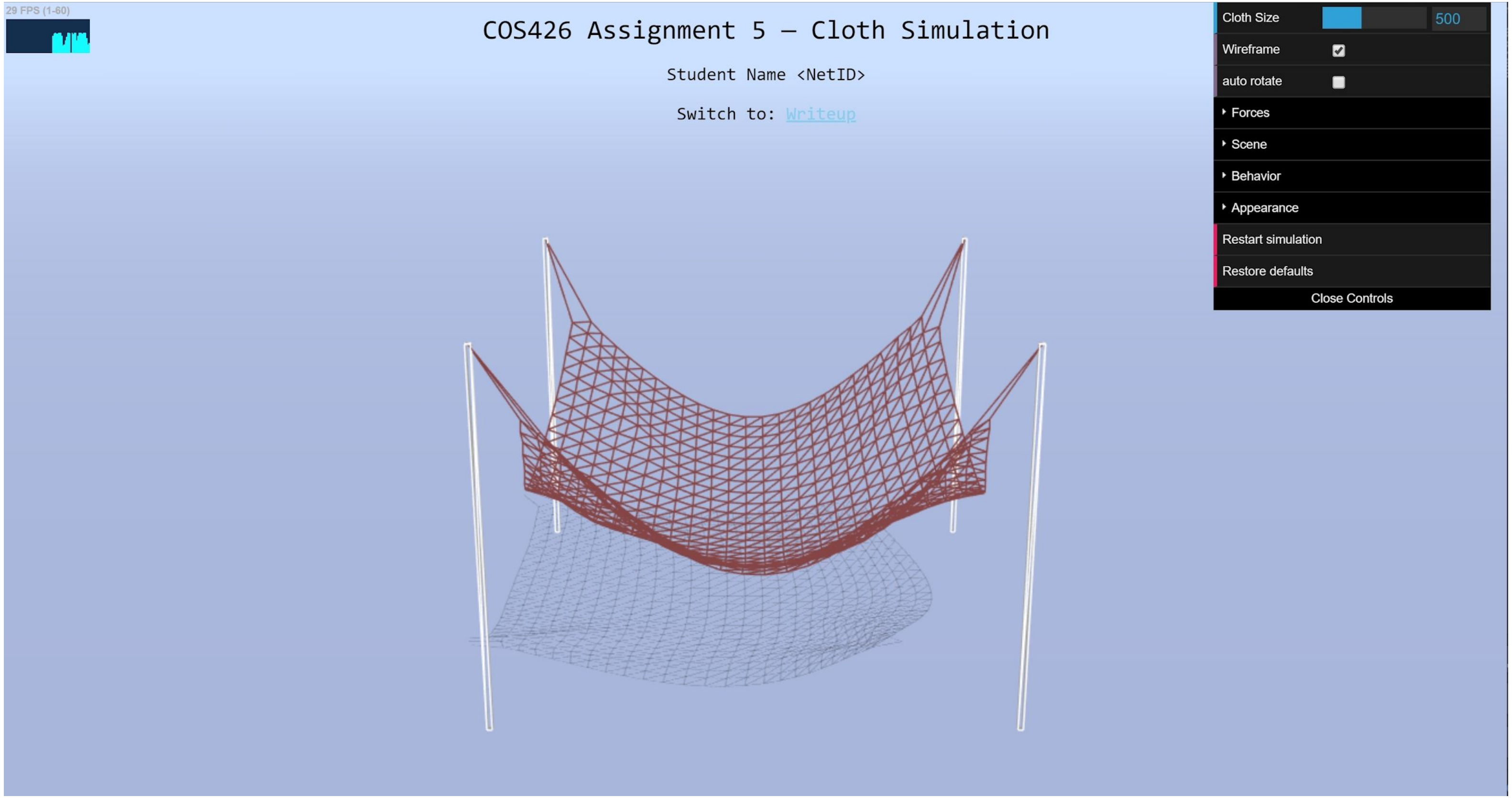
- Course Project
  - Groups of 2-4 strongly recommended
  - Stay tuned for more detailed spec
- Start thinking about ideas!
  - TAs are happy to provide early feedback
  - Former project "Hall of Fame" on course site
  - Or view all last year's submissions here

A5 Overview: Setup

Same as before:

- Run “`python3 -m http.server`” (or similar) inside the assignment directory
- Open “`http://localhost:8000`” in web browser

A5 Overview: GUI



A5 Overview: GUI

- Useful functions
  - Cloth size: change number of particles
  - Wireframe: change rendering style
  - Auto rotate: camera will orbit around scene
  - Wave: cloth oscillates up and down (useful debugging tool)
  - Appearance: change rendering properties
  - Image capture: ‘i’ to download a screenshot
  - Video capture: ‘v’ to start/stop recording

A5 Overview: GUI

- Features to implement:
  - **Events:** listen for and respond to user inputs
  - **Behavior:** model a cloth as a mass-and-spring system
  - **Forces:** apply and react to external forces and impulses
    - Gravity, wind, rain, …
  - **Scene:** collide with other objects in the scene

A5 Overview: Suggested Order

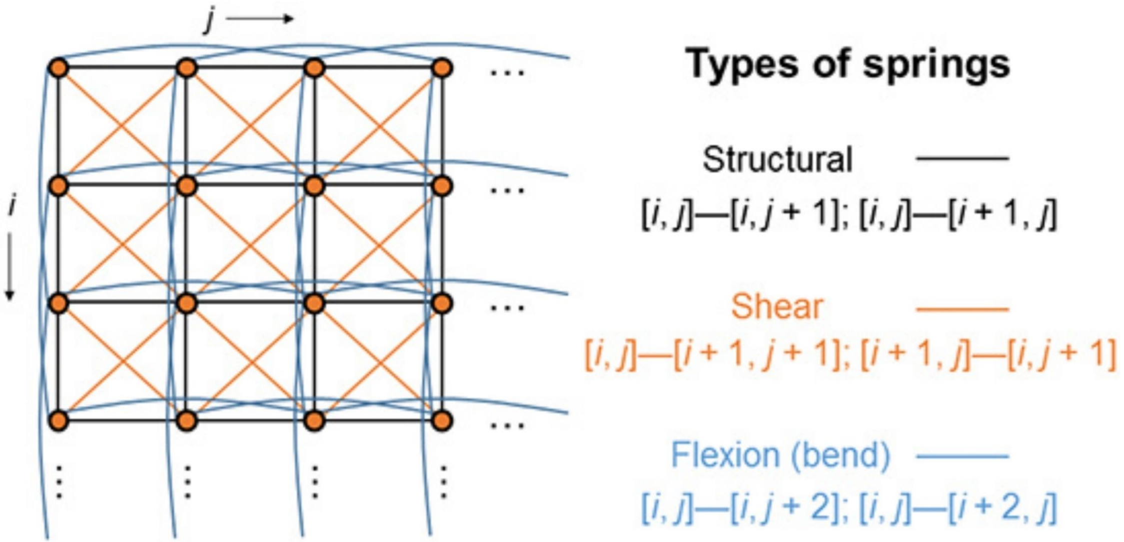
- First, implement **impact event handlers** for debugging
- Then, define & enforce **constraints**
  - Verify with your event handlers or the wave oscillator
- Move on to **forces** and **intersections** *only once these are working*

Physics-Based Cloth Simulation

- Represent cloth discretely as a grid of **point masses** connected by **springs**
- Each **point mass** is a single particle in the particle system
  - Each **point mass** is affected by **forces** in the system
- Each **spring** is a constraint on our particle system that holds the **point masses** together

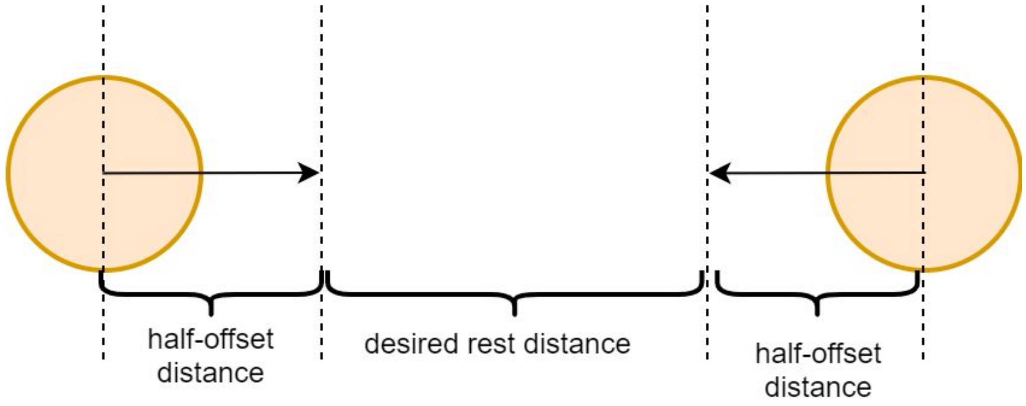
Three Types of Constraints

- Structural
  - 1-away neighbors in row and column
- Shear
  - 1-away neighbors diagonally
- Bending
  - 2-away neighbors in row and column



Enforcing Constraints

- Each **constraint** (spring) tries to keep the **particles** (point masses) on either end together at roughly their **natural rest distance**.
- At each timestep in the simulation, apply a “correction” directly to the position of both particles to bring them closer to their rest distance.



Simulation Loop

At a high level:

1. Accumulate forces acting on each particle (e.g. gravity)
2. Solve Newton's equations of motion (by numerical integration) to compute new positions for each particle
3. Handle collisions
4. Enforce constraints
5. Repeat from Step 1

Step 1: Accumulate Forces

- Each particle experiences some **net force** at every instant in time
- There are many possible forces
  - Gravity, wind, and so on...
  - For each particle, add up all force vectors acting on it into a single net force
- Each particle can also be affected by **spring forces** (Hooke's law) from nearby particles, but we omit this in A5

Step 2: Solve Equations of Motion

- **Numerically integrate position given v, a**
- Many choices are available:
  - Explicit Euler
  - Implicit Euler
  - **Verlet - good numerical stability, simple to implement**
  - Midpoint
  - Runge-Kutta
  - And more!

Step 2: Verlet Integration

- If we use a very small timestep **`dt`**, we can assume constant acceleration and velocity for the equations of motion
- Then, new position (at time **`t + dt`**) can be calculated the from old position (at time **`t`**):

$$x_{t+dt} = x_t + (1 - D) * v_t * dt + a_t * dt^2$$

- Note: $\mathbf{v_t}$ **`* dt`** is approximated by the change in position relative to the last timestep.
- **D** represents a constant damping factor in [0, 1].

Step 2: Timestep Tradeoffs

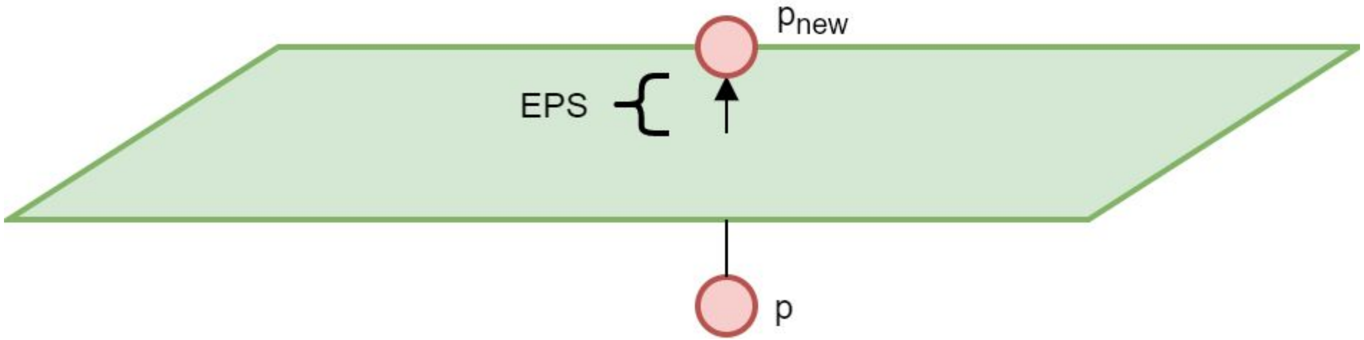
- Small timesteps provide greater stability and accuracy, but require more steps of the simulation (i.e. your simulation can be very slow) to achieve the same end results.
- Large timesteps will require less work and fewer steps of the simulation (i.e. your simulation will just run faster), but are prone to error
  - Timesteps that are too large may never find a “resting state”

Step 3: Handle Collisions

- Particles may collide with other objects (or even other particles in the same cloth!)
- Detect collisions in 3D space and apply a positional correction (easier to code) or a repelling force (more physically accurate)
  - In A5, we will apply a positional correction and simulate friction to still get visually plausible results

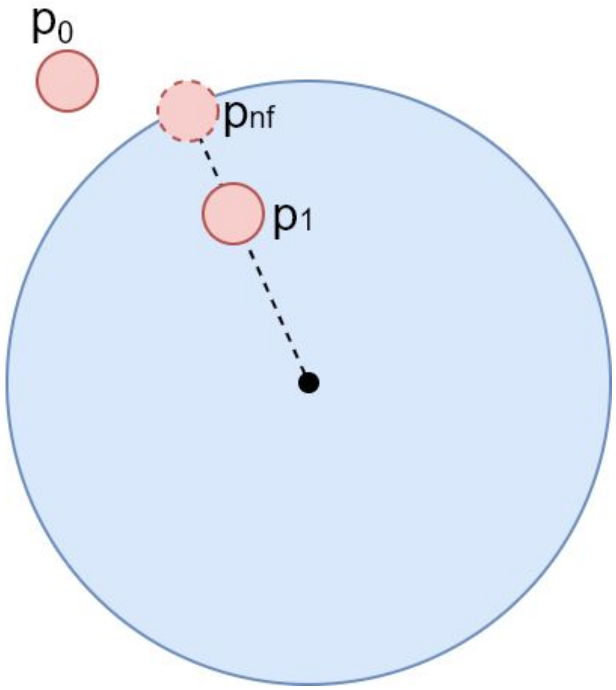
Step 3: Handle Collisions — Floor

- Assume infinite plane with cloth above it
  - Perform simple “hack” of pushing particle back to the surface of the floor if it goes under
  - Just like in A3, use EPS to ensure stability & avoid clipping



Step 3: Handle Collisions — Sphere

- Suppose that at time **`t`**,
  - the particle is just *barely outside* the sphere, at $\mathbf{p_0}$
- … and now at time **`t + dt`**,
  - the particle is just *barely inside* the sphere, at $\mathbf{p_1}$
  - There's been a **collision**!
- <u>If there is no friction,</u>
  - Project the particle's position to the closest point on the sphere's surface, called **`posNoFriction`** (p_{nf}).



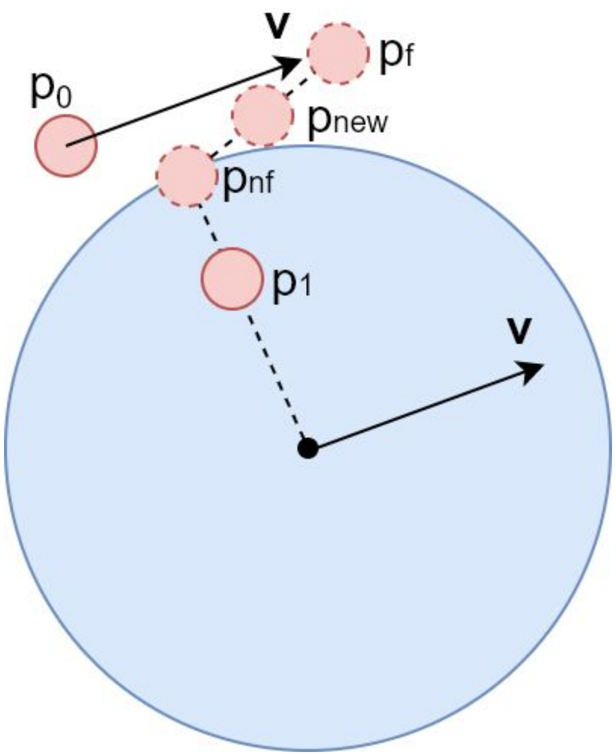
Step 3: Handle Collisions — Sphere

- If there is friction **F**,
  - then we want to simulate the particle “clinging onto” the sphere that it is in contact with, especially when it is moving.
- Adjust the particle’s previous position $\mathbf{p_0}$
  - … by the same motion **v** that the sphere made in the last timestep to get a new **posFriction** ($\mathbf{p_f}$)
- New particle position $\mathbf{p_{new}}$ is linearly interpolated:

```
newPos = [posFriction * F] + [posNoFriction * (1-F)]
```

Step 3: Handle Collisions — Sphere

- With friction:
  - Compute $\mathbf{p}_{nf}$ by projecting $\mathbf{p}_1$ onto the sphere
  - Compute $\mathbf{p}_f$ by adding to $\mathbf{p}_0$ the sphere's velocity $\mathbf{v}$
  - Compute $\mathbf{p}_{new}$ by linearly interpolating $\mathbf{p}_f$ & $\mathbf{p}_{nf}$



Step 3: Handle Collisions — Box

- Same idea as sphere!
  - Compute $\mathbf{p}_{nf}$ & $\mathbf{p}_f$ - then interpolate.
- Main difference:
  - find $\mathbf{p}_{nf}$ by projecting $\mathbf{p}_1$ onto the closest face of the box
- We set a `boundingBox` property on the box, which is a Three.js **`Box3`** object.
  - Consult the Box3 API!
  - You can use its **`min`** and **`max`** to help find the closest point on the box using some conditionals.
  - No need for complicated math!

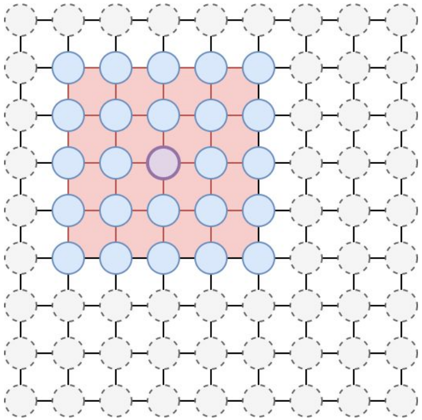
Step 3: Handle Collisions — Self

- Self-collision prevention is an optional feature
- Basic idea:
    - For each pair of particles in the cloth...
        - If they are too close (closer than rest distance), apply a correction shifting them both back towards the desired rest distance.
    - Very similar to how you enforce the constraints!
    - But the naive approach is slow...

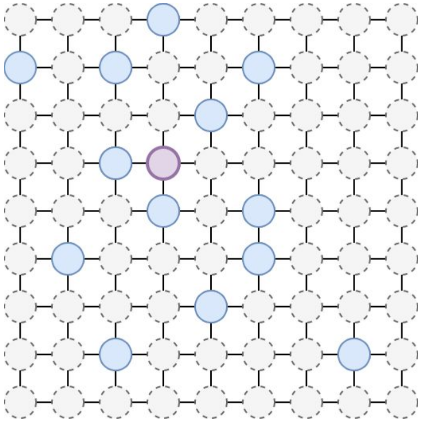
Step 3: Handle Collisions — Self

- Heuristic extensions:
  - Only enforce self-intersection constraints on some (possibly varying) subset of particle pairs at each timestep

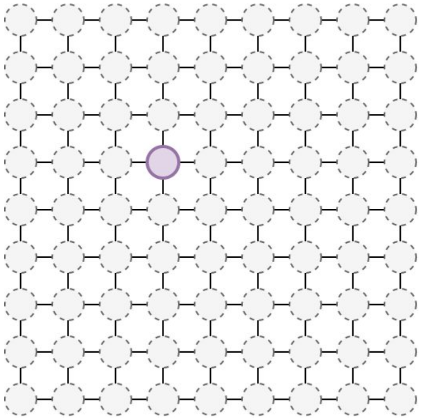
Perhaps by assuming 2D locality...



or by enforcing a random subset...

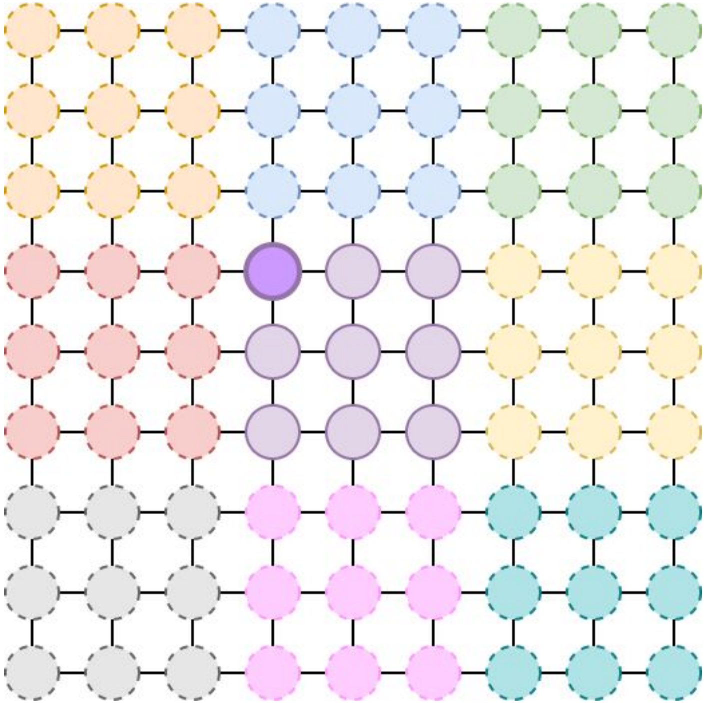


or an optimization of your own design...



Step 3: Handle Collisions — Self

- A complex (but more accurate) solution is **spatial hashing**.
- Place particles into bins based on their **current 3D position**, and only enforce constraints *within* each bin.
  - Bins may need to be *recomputed* after particles move.
  - Creating and assigning bins very similar to A3's *checkerboard material*
  - Use a *sparse* bin representation!
  - Corner cases require special care



Event Listeners

- Annoying we have no way to directly manipulate the cloth…
  - What if there were some way to move the cloth ourselves, just using keyboard and mouse?
- Your browser automatically captures keypresses, mouse movement, and tons of other **events**
  - By writing an **event listener**, we can register a callback for the browser to run any time one of these events is detected within a particular page element.
- Define an event listener to “bump” the cloth up/down or left/right when a certain key is pressed

Event Listeners: A simple example

- Event handlers are bound to a certain event type, like “*keyup*”, “*mousemove*”, or “*resize*”
- When that event occurs, all registered handlers are called with an **event object** containing the relevant parameters
    - Which key was pressed
    - The targeted page element
    - and so on…

```
// A simple keylogger
let keylog = function(event) {
    console.log(event.key);
}

window.addEventListener(
    “keydown”,
    keylog
);
```

Extensions

Extensions - Forces

- Time-varying, sinusoidal wind
  - ```
    s(t) = A[cos/sin](wt) + C
    ```
  - ```
    Wind = s(t) * <f(t), g(t), h(t)>
    ```
- Custom force
  - May vary as a function of space, time, and/or any other parameters you like!
  - Be creative: tractor beams, anti-gravity, or a black hole - the choice is yours!

Extensions - Forces

- Rain impulse
  - Model rainfall by simulating **sudden strikes** at random particles on the cloth.
  - An impulse, not a force - directly move particle positions in some *rainfall direction*
    - ...can be a constant, or varying with time/space
  - To model the physical size of a raindrop, apply a smaller offset to **nearby particles** as well

Extensions - Intersections

- General Plane Collisions
  - Floor collisions are a bit of a hack, reliant on the specifics of our scene.
  - Consider general plane equation **`dot(P,N) + D`**
  - Implement **collisions with plane** and account for friction
    - Very similar to intersecting with just one side of a box

Extensions - Scene

- New Objects
  - Add support for collisions with something other than a sphere, box, or plane
- Custom Scene
  - Put together an interesting scene in which a cloth interacts with multiple other objects
- Textures
  - Add your own textures to the scene, or use Three.js's libraries to support extra features, like normal mapping