IT303: Software engineering

Pandanvadara Sahil Rasikbhai
202201151
Lab 7

Git hub link: https://github.com/processing/p5.js/edit/main/src/webgl/3d_primitives.js#L261C0-L261C3

#1963 Line of code in parts

```
import p5 from '../core/main';
import './p5.Geometry';
import * as constants from '../core/constants';


 * Begins adding shapes to a new
 * <a href="#/p5.Geometry">p5.Geometry</a> object.
 *
 * The `beginGeometry()` and <a href="#/p5/endGeometry">endGeometry()</a>
 * functions help with creating complex 3D shapes from simpler ones such as
 * <a href="#/p5/sphere">sphere()</a>. `beginGeometry()` begins adding shapes
 * to a custom <a href="#/p5.Geometry">p5.Geometry</a> object and
 * <a href="#/p5/endGeometry">endGeometry()</a> stops adding them.
 *
 * `beginGeometry()` and <a href="#/p5/endGeometry">endGeometry()</a> can help
 * to make sketches more performant. For example, if a complex 3D shape
 * doesn't change while a sketch runs, then it can be created with
 * `beginGeometry()` and <a href="#/p5/endGeometry">endGeometry()</a>.
 * Creating a <a href="#/p5.Geometry">p5.Geometry</a> object once and then
 * drawing it will run faster than repeatedly drawing the individual pieces.
 *
 * See <a href="#/p5/buildGeometry">buildGeometry()</a> for another way to
 * build 3D shapes.
 *
 * Note: `beginGeometry()` can only be used in WebGL mode.
 *
 * @method beginGeometry
 *
 * @example
 * <div>
 * <code>
 * // Click and drag the mouse to view the scene from different angles.
 *
 * let shape;
 *
 * function setup() {
 *   createCanvas(100, 100, WEBGL);
```

```
*
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   // Add a cone.
*   cone();
*
*   // Stop building the p5.Geometry object.
*   shape = endGeometry();
*
*   describe('A white cone drawn on a gray background.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the p5.Geometry object.
*   noStroke();
*
*   // Draw the p5.Geometry object.
*   model(shape);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let shape;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the p5.Geometry object.
*   createArrow();
*
*   describe('A white arrow drawn on a gray background.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
```

```
*   orbit control();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the p5.Geometry object.
*   noStroke();
*
*   // Draw the p5.Geometry object.
*   model(shape);
* }
*
* function createArrow() {
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   // Add shapes.
*   push();
*   rotateX(PI);
*   cone(10);
*   translate(0, -10, 0);
*   cylinder(3, 20);
*   pop();
*
*   // Stop building the p5.Geometry object.
*   shape = endGeometry();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let blueArrow;
* let redArrow;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the arrows.
*   redArrow = createArrow('red');
*   blueArrow = createArrow('blue');
*
*   describe('A red arrow and a blue arrow drawn on a gray background. The blue arrow rotates slowly.');
* }
*
* function draw() {
*   background(200);
*
```

```
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the arrows.
*   noStroke();
*
*   // Draw the red arrow.
*   model(redArrow);
*
*   // Translate and rotate the coordinate system.
*   translate(30, 0, 0);
*   rotateZ(frameCount * 0.01);
*
*   // Draw the blue arrow.
*   model(blueArrow);
* }
*
* function createArrow(fillColor) {
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   fill(fillColor);
*
*   // Add shapes to the p5.Geometry object.
*   push();
*   rotateX(PI);
*   cone(10);
*   translate(0, -10, 0);
*   cylinder(3, 20);
*   pop();
*
*   // Stop building the p5.Geometry object.
*   let shape = endGeometry();
*
*   return shape;
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let button;
* let particles;
*
* function setup() {
```

```
*   createCanvas(100, 100, WEBGL);
*
*   // Create a button to reset the particle system.
*   button = createButton('Reset');
*
*   // Call resetModel() when the user presses the button.
*   button.mousePressed(resetModel);
*
*   // Add the original set of particles.
*   resetModel();
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the particles.
*   noStroke();
*
*   // Draw the particles.
*   model(particles);
* }
*
* function resetModel() {
*   // If the p5.Geometry object has already been created,
*   // free those resources.
*   if (particles) {
*     freeGeometry(particles);
*   }
*
*   // Create a new p5.Geometry object with random spheres.
*   particles = createParticles();
* }
*
* function createParticles() {
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   // Add shapes.
*   for (let i = 0; i < 60; i += 1) {
*     // Calculate random coordinates.
*     let x = randomGaussian(0, 20);
*     let y = randomGaussian(0, 20);
*     let z = randomGaussian(0, 20);
*
```

```
 *    push();
 *    // Translate to the particle's coordinates.
 *    translate(x, y, z);
 *    // Draw the particle.
 *    sphere(5);
 *    pop();
 *  }
 *
 *  // Stop building the p5.Geometry object.
 *  let shape = endGeometry();
 *
 *  return shape;
 * }
 * </code>
 * </div>

p5.prototype.beginGeometry = function() {
  return this._renderer.beginGeometry();
};
```

1. How many errors can you identify in the program? List the errors below.

Category A: Data Reference Errors

Uninitialised or Unset Variables: In the example code snippets, variables such as `shape`, `redArrow`, `blueArrow`, and `particles` are initialised correctly. However, there are potential risks if these variables are used before initialisation. For example, the `resetModel()` function does check if `particles` are defined before calling `freeGeometry()`, which is good practice.

Category C: Computation Errors

Division by Zero Risk: There are no explicit divisions in this program; hence, there isn't a direct risk of a division by zero error.

Category D: Comparison Errors

Faulty Comparison Logic: No faulty comparisons are present in the given code. The program uses conditions for control flow, but these don't involve problematic comparisons.

Category E: Control-Flow Errors

Off-by-One Mistake: There aren't any noticeable off-by-one errors in the given loops or control structures.

Category F: Interface Errors

Incorrect Handling of Parameters: The program should validate input formats and parameters, such as ensuring the types and values are suitable for the intended operations. For instance, when creating the arrows, the `fillColor` parameter should be validated before applying it.

Category G: Input/Output Errors

Missing Error Handling for Input Data: There is no error handling for potential issues with the input data. The code should include checks to ensure that the parameters passed to functions like `cone()` and `cylinder()` are valid numbers.

Category H: Other Checks

Missing Libraries: The code snippets are built around the `p5` library for JavaScript, but the user needs to ensure that `p5.js` and any relevant plugins (such as `p5.Geometry`) are correctly imported.

2. Which category of inspection would you consider more useful?

Category A: Data Reference Errors

- Ensuring variables like `shape`, `redArrow`, and `particles` are properly initialized is crucial. Uninitialized variables can cause errors or render nothing, making data reference checks more relevant for this code.

3. Which type of error is not identifiable using program inspection?

Logic Errors

- Program inspection can miss subtle logic issues where the code runs without errors but produces incorrect results, such as misplaced shapes or unexpected rendering behavior.

4. Is program inspection worth applying?

Yes.

- It helps catch common issues like uninitialized variables or control-flow mistakes. However, it should be paired with unit tests, integration tests, and visual checks to catch logic errors in 3D rendering.

```
 * Stops adding shapes to a new
 * <a href="#/p5.Geometry">p5.Geometry</a> object and returns the object.
 *
```

```
* The `beginGeometry()` and <a href="#/p5/endGeometry">endGeometry()</a>
* functions help with creating complex 3D shapes from simpler ones such as
* <a href="#/p5/sphere">sphere()</a>. `beginGeometry()` begins adding shapes
* to a custom <a href="#/p5.Geometry">p5.Geometry</a> object and
* <a href="#/p5/endGeometry">endGeometry()</a> stops adding them.
*
* `beginGeometry()` and <a href="#/p5/endGeometry">endGeometry()</a> can help
* to make sketches more performant. For example, if a complex 3D shape
* doesn't change while a sketch runs, then it can be created with
* `beginGeometry()` and <a href="#/p5/endGeometry">endGeometry()</a>.
* Creating a <a href="#/p5.Geometry">p5.Geometry</a> object once and then
* drawing it will run faster than repeatedly drawing the individual pieces.
*
* See <a href="#/p5/buildGeometry">buildGeometry()</a> for another way to
* build 3D shapes.
*
* Note: `endGeometry()` can only be used in WebGL mode.
*
* @method endGeometry
* @returns {p5.Geometry} new 3D shape.
*
* @example
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let shape;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   // Add a cone.
*   cone();
*
*   // Stop building the p5.Geometry object.
*   shape = endGeometry();
*
*   describe('A white cone drawn on a gray background.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
```

```
*   lights();
*
*   // Style the p5.Geometry object.
*   noStroke();
*
*   // Draw the p5.Geometry object.
*   model(shape);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let shape;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the p5.Geometry object.
*   createArrow();
*
*   describe('A white arrow drawn on a gray background.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the p5.Geometry object.
*   noStroke();
*
*   // Draw the p5.Geometry object.
*   model(shape);
* }
*
* function createArrow() {
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   // Add shapes.
*   push();
*   rotateX(PI);
*   cone(10);
```

```
*   translate(0, -10, 0);
*   cylinder(3, 20);
*   pop();
*
*   // Stop building the p5.Geometry object.
*   shape = endGeometry();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let blueArrow;
* let redArrow;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the arrows.
*   redArrow = createArrow('red');
*   blueArrow = createArrow('blue');
*
*   describe('A red arrow and a blue arrow drawn on a gray background. The blue arrow rotates slowly.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the arrows.
*   noStroke();
*
*   // Draw the red arrow.
*   model(redArrow);
*
*   // Translate and rotate the coordinate system.
*   translate(30, 0, 0);
*   rotateZ(frameCount * 0.01);
*
*   // Draw the blue arrow.
*   model(blueArrow);
* }
*
```

```
* function createArrow(fillColor) {
*   // Start building the p5.Geometry object.
*   beginGeometry();
*
*   fill(fillColor);
*
*   // Add shapes to the p5.Geometry object.
*   push();
*   rotateX(PI);
*   cone(10);
*   translate(0, -10, 0);
*   cylinder(3, 20);
*   pop();
*
*   // Stop building the p5.Geometry object.
*   let shape = endGeometry();
*
*   return shape;
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let button;
* let particles;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create a button to reset the particle system.
*   button = createButton('Reset');
*
*   // Call resetModel() when the user presses the button.
*   button.mousePressed(resetModel);
*
*   // Add the original set of particles.
*   resetModel();
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
```

```
 *
 *   // Style the particles.
 *   noStroke();
 *
 *   // Draw the particles.
 *   model(particles);
 * }
 *
 * function resetModel() {
 *   // If the p5.Geometry object has already been created,
 *   // free those resources.
 *   if (particles) {
 *     freeGeometry(particles);
 *   }
 *
 *   // Create a new p5.Geometry object with random spheres.
 *   particles = createParticles();
 * }
 *
 * function createParticles() {
 *   // Start building the p5.Geometry object.
 *   beginGeometry();
 *
 *   // Add shapes.
 *   for (let i = 0; i < 60; i += 1) {
 *     // Calculate random coordinates.
 *     let x = randomGaussian(0, 20);
 *     let y = randomGaussian(0, 20);
 *     let z = randomGaussian(0, 20);
 *
 *     push();
 *     // Translate to the particle's coordinates.
 *     translate(x, y, z);
 *     // Draw the particle.
 *     sphere(5);
 *     pop();
 *   }
 *
 *   // Stop building the p5.Geometry object.
 *   let shape = endGeometry();
 *
 *   return shape;
 * }
 * </code>
 * </div>
 */
p5.prototype.endGeometry = function() {
  return this._renderer.endGeometry();
};
```

1. How many errors are there in the program? Mention the errors you have identified.

1. Identify Errors in the Code:

Category A: Data Reference Errors

- Uninitialized or Incorrect Variable Use: In multiple examples, the `shape`, `redArrow`, `blueArrow`, and `particles` variables must be initialized correctly before being used. Using them without setting the values can cause runtime errors.

Category C: Computation Errors

- Transformation Issues: If transformations (like `translate()`, `rotate()`) are not correctly reset (e.g., missing `push()`/`pop()`), they could apply unintended effects on other objects.

Category F: Interface Errors

- Parameter Handling: Some functions, such as `createArrow()`, should validate input parameters like `fillColor`. If invalid values are passed, the function should handle the issue gracefully.

Category G: Input/Output Errors

- Error Handling for User Actions: The button actions and user inputs should be validated. For instance, checking if the `particles` object is indeed initialized before trying to free it.

2. Which category of inspection would you consider more useful?

Category A: Data Reference Errors

- Given the nature of this code, ensuring that variables (like `shape`, `particles`) are properly initialized before use is crucial to avoid runtime issues. For 3D rendering, managing these data references is essential for stable execution.

3. Which type of error are you not able to identify using the program inspection?

Logic Errors

- Program inspection may not catch visual issues or subtle rendering logic problems, such as incorrect placement of 3D shapes. These errors would manifest as visual artifacts, and their detection requires actual rendering and visual inspection.

4. Is Program Inspection Worth Applying?

Yes.

- It helps catch uninitialized variables, improper control flow, or incorrect use of transformations. However, it should be used alongside other testing methods like unit tests, integration tests, and visual inspection to ensure 3D graphics render correctly.

```
* Creates a custom <a href="#/p5.Geometry">p5.Geometry</a> object from
* simpler 3D shapes.
*
* `buildGeometry()` helps with creating complex 3D shapes from simpler ones
* such as <a href="#/p5/sphere">sphere()</a>. It can help to make sketches
* more performant. For example, if a complex 3D shape doesn't change while a
* sketch runs, then it can be created with `buildGeometry()`. Creating a
* <a href="#/p5.Geometry">p5.Geometry</a> object once and then drawing it
* will run faster than repeatedly drawing the individual pieces.
*
* The parameter, `callback`, is a function with the drawing instructions for
* the new <a href="#/p5.Geometry">p5.Geometry</a> object. It will be called
* once to create the new 3D shape.
*
* See <a href="#/p5/beginGeometry">beginGeometry()</a> and
* <a href="#/p5/endGeometry">endGeometry()</a> for another way to build 3D
* shapes.
*
* Note: `buildGeometry()` can only be used in WebGL mode.
*
* @method buildGeometry
* @param {Function} callback function that draws the shape.
* @returns {p5.Geometry} new 3D shape.
*
* @example
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let shape;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the p5.Geometry object.
*   shape = buildGeometry(createShape);
*
*   describe('A white cone drawn on a gray background.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
```

```
*
*   // Style the p5.Geometry object.
*   noStroke();
*
*   // Draw the p5.Geometry object.
*   model(shape);
* }
*
* // Create p5.Geometry object from a single cone.
* function createShape() {
*   cone();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let shape;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the arrow.
*   shape = buildGeometry(createArrow);
*
*   describe('A white arrow drawn on a gray background.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the arrow.
*   noStroke();
*
*   // Draw the arrow.
*   model(shape);
* }
*
* function createArrow() {
*   // Add shapes to the p5.Geometry object.
*   push();
*   rotateX(PI);
```

```
*   cone(10);
*   translate(0, -10, 0);
*   cylinder(3, 20);
*   pop();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let shape;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create the p5.Geometry object.
*   shape = buildGeometry(createArrow);
*
*   describe('Two white arrows drawn on a gray background. The arrow on the right rotates slowly.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the arrows.
*   noStroke();
*
*   // Draw the p5.Geometry object.
*   model(shape);
*
*   // Translate and rotate the coordinate system.
*   translate(30, 0, 0);
*   rotateZ(frameCount * 0.01);
*
*   // Draw the p5.Geometry object again.
*   model(shape);
* }
*
* function createArrow() {
*   // Add shapes to the p5.Geometry object.
*   push();
*   rotateX(PI);
```

```
*   cone(10);
*   translate(0, -10, 0);
*   cylinder(3, 20);
*   pop();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* let button;
* let particles;
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   // Create a button to reset the particle system.
*   button = createButton('Reset');
*
*   // Call resetModel() when the user presses the button.
*   button.mousePressed(resetModel);
*
*   // Add the original set of particles.
*   resetModel();
*
*   describe('A set of white spheres on a gray background. The spheres are positioned randomly. Their positions reset when the user presses the Reset button.');
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the particles.
*   noStroke();
*
*   // Draw the particles.
*   model(particles);
* }
*
* function resetModel() {
*   // If the p5.Geometry object has already been created,
*   // free those resources.
```

```
 *   if (particles) {
 *     freeGeometry(particles);
 *   }
 *
 *   // Create a new p5.Geometry object with random spheres.
 *   particles = buildGeometry(createParticles);
 * }
 *
 * function createParticles() {
 *   for (let i = 0; i < 60; i += 1) {
 *     // Calculate random coordinates.
 *     let x = randomGaussian(0, 20);
 *     let y = randomGaussian(0, 20);
 *     let z = randomGaussian(0, 20);
 *
 *     push();
 *     // Translate to the particle's coordinates.
 *     translate(x, y, z);
 *     // Draw the particle.
 *     sphere(5);
 *     pop();
 *   }
 * }
 * </code>
 * </div>
 */
p5.prototype.buildGeometry = function(callback) {
  return this._renderer.buildGeometry(callback);
};
```

1. How many errors are there in the program? Mention the errors you have identified.

1. Identify Errors in the Code:

Category A: Data Reference Errors

- Uninitialized Variables: Variables like `shape` and `particles` need to be properly initialized before they are used. Failing to do this may result in runtime errors when trying to render them.

Category C: Computation Errors

- Transformation State Management: If transformations (such as `translate()`, `rotate()`) are used without proper state management using `push()` and `pop()`, unintended side effects on other objects can occur.

Category G: Input/Output Errors

- Callback Handling: The `callback` parameter should be validated before being used in `buildGeometry()`. If an invalid callback function is provided, it should handle the situation gracefully to avoid runtime issues.

2. Which Category of Inspection is More Useful?

Category A: Data Reference Errors

- Ensuring that variables like `shape` and `particles` are correctly initialized and managed is crucial, as 3D rendering heavily relies on properly defined objects to avoid rendering issues and crashes.

3. Which Type of Error is Not Identifiable Using Program Inspection?

Logic Errors

- Visual rendering issues, such as incorrect placement or unintended transformations of 3D objects, might not be evident through program inspection alone. These kinds of errors are best detected through visual testing of the rendered output.

4. Is Program Inspection Worth Applying?

Yes.Program inspection is valuable for identifying uninitialized variables, transformation issues, and other common programming mistakes. However, it should be supplemented with other techniques, like visual testing, to ensure the correctness of 3D rendering and transformations.

* Clears a p5.Geometry object from the graphics

* processing unit (GPU) memory.

*

* p5.Geometry objects can contain lots of data

* about their vertices, surface normals, colors, and so on. Complex 3D shapes

* can use lots of memory which is a limited resource in many GPUs. Calling

* `freeGeometry()` can improve performance by freeing a

* p5.Geometry object's resources from GPU memory.

* `freeGeometry()` works with p5.Geometry objects

* created with beginGeometry() and

* endGeometry(),

* buildGeometry(), and

* loadModel().

*

* The parameter, `geometry`, is the p5.Geometry

* object to be freed.

*

* Note: A p5.Geometry object can still be drawn

* after its resources are cleared from GPU memory. It may take longer to draw

* the first time it's redrawn.

*

* Note: `freeGeometry()` can only be used in WebGL mode.

*

* @method freeGeometry

* @param {p5.Geometry} geometry 3D shape whose resources should be freed.

*

* @example

*

* <code>

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   background(200);

*

*   // Create a p5.Geometry object.

*   beginGeometry();

*   cone();

*   let shape = endGeometry();

*

*   // Draw the shape.

*   model(shape);

*

*   // Free the shape's resources.

*   freeGeometry(shape);

* }

* </code>

*

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* let button;

* let particles;

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   // Create a button to reset the particle system.

```
*   button = createButton('Reset');
*
*   // Call resetModel() when the user presses the button.
*   button.mousePressed(resetModel);
*
*   // Add the original set of particles.
*   resetModel();
* }
*
* function draw() {
*   background(50);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Turn on the lights.
*   lights();
*
*   // Style the particles.
*   noStroke();
*
*   // Draw the particles.
*   model(particles);
* }
*
* function resetModel() {
```

```
*   // If the p5.Geometry object has already been created,
*   // free those resources.
*   if (particles) {
*     freeGeometry(particles);
*   }
*
*   // Create a new p5.Geometry object with random spheres.
*   particles = buildGeometry(createParticles);
* }
*
* function createParticles() {
*   for (let i = 0; i < 60; i += 1) {
*     // Calculate random coordinates.
*     let x = randomGaussian(0, 20);
*     let y = randomGaussian(0, 20);
*     let z = randomGaussian(0, 20);
*
*     push();
*     // Translate to the particle's coordinates.
*     translate(x, y, z);
*     // Draw the particle.
*     sphere(5);
*     pop();
*   }
* }
* </code>
```

```
 * </div>

 */

p5.prototype.freeGeometry = function(geometry) {

  this._renderer._freeBuffers(geometry.gid);

};
```

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

- Null or Undefined References: The code should check if the `geometry` parameter is valid (i.e., not null or undefined) before attempting to free its resources. If `geometry` is not valid, it could lead to runtime errors when calling `freeGeometry()`.

Category C: Computation Errors

- State Management: Freeing a `p5.Geometry` object too soon or without checking if it has already been freed could lead to unexpected behaviors when trying to draw or manipulate the object afterward.

Category G: Input/Output Errors

- Invalid Input Handling: The function should validate that the provided argument is indeed a `p5.Geometry` object before trying to release its resources.

2. Which Category of Inspection is More Useful?

Category A: Data Reference Errors

- Ensuring that the `geometry` object is valid before performing operations on it is crucial. It helps avoid potential crashes or unexpected behavior in the code.

3. Which Type of Error is Not Identifiable Using Program Inspection?

Logic Errors

- Issues such as freeing a `p5.Geometry` object while it is still in use or failing to manage GPU resources effectively may not be caught through inspection alone. These types of errors require runtime testing to identify.

4. Is Program Inspection Worth Applying?

Yes.

- Program inspection is valuable for spotting potential issues with uninitialized variables, null references, and incorrect handling of object states. However, it should be combined with testing methods like runtime testing and visual verification to catch more subtle logic errors related to the correct freeing and use of GPU resources.

* //Draws a plane.

*

* //A plane is a four-sided, flat shape with every angle measuring 90˚. It's

* //similar to a rectangle and offers advanced drawing features in WebGL mode.

*

* //The first parameter, `width`, is optional. If a `Number` is passed, as in

*// `plane(20)`, it sets the plane's width and height. By default, `width` is

*// 50.

*

* //The second parameter, `height`, is also optional. If a `Number` is passed,

*//as in `plane(20, 30)`, it sets the plane's height. By default, `height` is

* //set to the plane's `width`.

*

*// The third parameter, `detailX`, is also optional. If a `Number` is passed,

*// as in `plane(20, 30, 5)` it sets the number of triangle subdivisions to use

*//along the x-axis. All 3D shapes are made by connecting triangles to form

*// their surfaces. By default, `detailX` is 1.

*

* //The fourth parameter, `detailY`, is also optional. If a `Number` is passed,

* //as in `plane(20, 30, 5, 7)` it sets the number of triangle subdivisions to

* //use along the y-axis. All 3D shapes are made by connecting triangles to

*// form their surfaces. By default, `detailY` is 1.

```
*
*// Note: `plane()` can only be used in WebGL mode.
*
* @method plane
* @param  {Number} [width]    width of the plane.
* @param  {Number} [height]   height of the plane.
* @param  {Integer} [detailX] number of triangle subdivisions along the x-axis.
* @param {Integer} [detailY]  number of triangle subdivisions along the y-axis.
* @chainable
*
* @example
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white plane on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
```

*

*   // Draw the plane.

*   plane();

* }

* </code>

*

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white plane on a gray background.');

* }

*

* function draw() {

*   background(200);

*

*   // Enable orbiting with the mouse.

*   orbitControl();

*

*   // Draw the plane.

*   // Set its width and height to 30.

*   plane(30);

```
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white plane on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the plane.
*   // Set its width to 30 and height to 50.
*   plane(30, 50);
* }
* </code>
* </div>
```

```
 */
p5.prototype.plane = function(
  width = 50,
  height = width,
  detailX = 1,
  detailY = 1
) {
  this._assert3d('plane');
  p5._validateParameters('plane', arguments);

  const gId = `plane|${detailX}|${detailY}`;

  if (!this._renderer.geometryInHash(gId)) {
    const _plane = function() {
      let u, v, p;
      for (let i = 0; i <= this.detailY; i++) {
        v = i / this.detailY;
        for (let j = 0; j <= this.detailX; j++) {
          u = j / this.detailX;
          p = new p5.Vector(u - 0.5, v - 0.5, 0);
          this.vertices.push(p);
          this.uvs.push(u, v);
        }
      }
    };
    const planeGeom = new p5.Geometry(detailX, detailY, _plane);
```

```
    planeGeom.computeFaces().computeNormals();
    if (detailX <= 1 && detailY <= 1) {
      planeGeom._makeTriangleEdges()._edgesToVertices();
    } else if (this._renderer._doStroke) {
      console.log(
        'Cannot draw stroke on plane objects with more' +
        ' than 1 detailX or 1 detailY'
      );
    }
    this._renderer.createBuffers(gId, planeGeom);
  }

  this._renderer.drawBuffersScaled(gId, width, height, 1);
  return this;
};
```

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

Uninitialized or Null References: The function does not seem to check whether the `width`, `height`, `detailX`, or `detailY` parameters are valid. If invalid values (like null or non-numeric types) are passed, it could cause unexpected behavior.

Category B: Data Handling Errors

Default Parameter Handling: The function defaults `height` to the value of `width`. While this works in most cases, passing in null or other non-number values could lead to unpredictable behavior.

Category C: Computation Errors

Geometry Hash Checking and Creation: The function uses the `gId` to determine if a geometry has already been created and cached. If this mechanism fails, it could lead to unnecessary recalculations or memory use.

2. Which Category of Inspection is More Useful?

Category A: Data Reference Errors

Ensuring that the input parameters are valid before using them is essential to prevent runtime issues and unexpected behavior.

3. Which Type of Error is Not Identifiable Using Program Inspection?

Visual or Rendering Issues

While inspecting the code can ensure logical correctness, it cannot guarantee that the plane is rendered as expected on the screen. This requires visual testing or runtime testing to verify.

4. Is Program Inspection Worth Applying?

Yes.

Inspection can reveal issues related to input validation, initialization, and the handling of cached geometries. It can help identify potential areas for optimization and error handling, ensuring better performance and fewer bugs.

```
/**
 * Draws a box (rectangular prism).
 *
 * A box is a 3D shape with six faces. Each face makes a 90˚ with four
 * neighboring faces.
 *
 * The first parameter, `width`, is optional. If a `Number` is passed, as in
 * `box(20)`, it sets the box's width and height. By default, `width` is 50.
 *
 * The second parameter, `height`, is also optional. If a `Number` is passed,
 * as in `box(20, 30)`, it sets the box's height. By default, `height` is set
 * to the box's `width`.
 *
 * The third parameter, `depth`, is also optional. If a `Number` is passed, as
```

```
* in `box(20, 30, 40)`, it sets the box's depth. By default, `depth` is set
* to the box's `height`.
*
* The fourth parameter, `detailX`, is also optional. If a `Number` is passed,
* as in `box(20, 30, 40, 5)`, it sets the number of triangle subdivisions to
* use along the x-axis. All 3D shapes are made by connecting triangles to
* form their surfaces. By default, `detailX` is 1.
*
* The fifth parameter, `detailY`, is also optional. If a number is passed, as
* in `box(20, 30, 40, 5, 7)`, it sets the number of triangle subdivisions to
* use along the y-axis. All 3D shapes are made by connecting triangles to
* form their surfaces. By default, `detailY` is 1.
*
* Note: `box()` can only be used in WebGL mode.
*
* @method  box
* @param  {Number} [width]     width of the box.
* @param  {Number} [height]    height of the box.
* @param  {Number} [depth]     depth of the box.
* @param {Integer} [detailX]   number of triangle subdivisions along the x-axis.
* @param {Integer} [detailY]   number of triangle subdivisions along the y-axis.
* @chainable
**/
* @example
* <div>
* <code>
```

```
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white box on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the box.
*   box();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
```

```
*
*   describe('A white box on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the box.
*   // Set its width and height to 30.
*   box(30);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white box on a gray background.');
* }
```

```
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the box.
*   // Set its width to 30 and height to 50.
*   box(30, 50);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white box on a gray background.');
* }
*
* function draw() {
*   background(200);
```

```
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the box.
*   // Set its width to 30, height to 50, and depth to 10.
*   box(30, 50, 10);
* }
* </code>
* </div>
*/
p5.prototype.box = function(width, height, depth, detailX, detailY) {
  this._assert3d('box');
  p5._validateParameters('box', arguments);
  if (typeof width === 'undefined') {
    width = 50;
  }
  if (typeof height === 'undefined') {
    height = width;
  }
  if (typeof depth === 'undefined') {
    depth = height;
  }

  const perPixelLighting =
    this._renderer.attributes && this._renderer.attributes.perPixelLighting;
```

```
if (typeof detailX === 'undefined') {
  detailX = perPixelLighting ? 1 : 4;
}
if (typeof detailY === 'undefined') {
  detailY = perPixelLighting ? 1 : 4;
}

const gId = `box|${detailX}|${detailY}`;
if (!this._renderer.geometryInHash(gId)) {
  const _box = function() {
    const cubeIndices = [
      [0, 4, 2, 6], // -1, 0, 0],// -x
      [1, 3, 5, 7], // +1, 0, 0],// +x
      [0, 1, 4, 5], // 0, -1, 0],// -y
      [2, 6, 3, 7], // 0, +1, 0],// +y
      [0, 2, 1, 3], // 0, 0, -1],// -z
      [4, 5, 6, 7] // 0, 0, +1] // +z
    ];
    //using custom edges
    //to avoid diagonal stroke lines across face of box
    this.edges = [
      [0, 1],
      [1, 3],
      [3, 2],
      [6, 7],
      [8, 9],
```

```
  [9, 11],
  [14, 15],
  [16, 17],
  [17, 19],
  [18, 19],
  [20, 21],
  [22, 23]
];

cubeIndices.forEach((cubeIndex, i) => {
  const v = i * 4;
  for (let j = 0; j < 4; j++) {
    const d = cubeIndex[j];
    //inspired by lightgl:
    //https://github.com/evanw/lightgl.js
    //octants:https://en.wikipedia.org/wiki/Octant_(solid_geometry)
    const octant = new p5.Vector(
      ((d & 1) * 2 - 1) / 2,
      ((d & 2) - 1) / 2,
      ((d & 4) / 2 - 1) / 2
    );
    this.vertices.push(octant);
    this.uvs.push(j & 1, (j & 2) / 2);
  }
  this.faces.push([v, v + 1, v + 2]);
  this.faces.push([v + 2, v + 1, v + 3]);
```

```
    });
  };
  const boxGeom = new p5.Geometry(detailX, detailY, _box);
  boxGeom.computeNormals();
  if (detailX <= 4 && detailY <= 4) {
    boxGeom._edgesToVertices();
  } else if (this._renderer._doStroke) {
    console.log(
      'Cannot draw stroke on box objects with more' +
      ' than 4 detailX or 4 detailY'
    );
  }
  //initialize our geometry buffer with
  //the key val pair:
  //geometry Id, Geom object
  this._renderer.createBuffers(gId, boxGeom);
 }
 this._renderer.drawBuffersScaled(gId, width, height, depth);

 return this;
};
```

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

- **Uninitialized or Null References:** The code does handle some uninitialized values for parameters (`width`, `height`, `depth`) by setting defaults if they are `undefined`. However, it does not validate that they are positive numbers, which could lead to visual issues.

Category B: Data Handling Errors

- **Handling of Detail Parameters (`detailX`, `detailY`):** The code has a mechanism for setting default values of `detailX` and `detailY` based on the lighting settings. However, if `detailX` or `detailY` are passed as invalid values, this could still cause problems.

Category C: Computation Errors

- **Geometry Hashing and Buffer Creation:** The `gId` is used to determine if a cached geometry exists, preventing unnecessary recalculations. If this check fails or is implemented incorrectly, it could lead to higher memory usage or redundant calculations.
- **Edge Handling for Stroke:** There is a check for drawing strokes on boxes with detail values above 4. If this limit is not respected, it might result in visual artifacts or improper rendering.

2. Which Category of Inspection is More Useful?

Category A: Data Reference Errors

- Ensuring that the inputs are valid (e.g., positive numbers) is essential to prevent rendering problems and improve the robustness of the function.

3. Which Type of Error is Not Identifiable Using Program Inspection?

Rendering Issues or Visual Artifacts

- While code inspection ensures logical correctness, it does not verify how the box appears visually in different scenarios. Visual testing is necessary to identify such issues.

4. Is Program Inspection Worth Applying?

Yes.

- Program inspection is beneficial to uncover problems related to input handling, geometry caching, and buffer management. It can also help optimize the code to ensure better memory and performance management.

* Draws a sphere.

*

* A sphere is a 3D shape with triangular faces that connect to form a round

* surface. Spheres with few faces look like crystals. Spheres with many faces

* have smooth surfaces and look like balls.

*

* The first parameter, `radius`, is optional. If a `Number` is passed, as in

* `sphere(20)`, it sets the radius of the sphere. By default, `radius` is 50.

*

* The second parameter, `detailX`, is also optional. If a `Number` is passed,

* as in `sphere(20, 5)`, it sets the number of triangle subdivisions to use

* along the x-axis. All 3D shapes are made by connecting triangles to form

* their surfaces. By default, `detailX` is 24.

*

* The third parameter, `detailY`, is also optional. If a `Number` is passed,

* as in `sphere(20, 5, 2)`, it sets the number of triangle subdivisions to

* use along the y-axis. All 3D shapes are made by connecting triangles to

* form their surfaces. By default, `detailY` is 16.

*

* Note: `sphere()` can only be used in WebGL mode.

*

* @method sphere

* @param {Number} [radius]   radius of the sphere. Defaults to 50.

* @param {Integer} [detailX] number of triangle subdivisions along the x-axis. Defaults to 24.

* @param {Integer} [detailY] number of triangle subdivisions along the y-axis. Defaults to 16.

*

* @chainable

* @example

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white sphere on a gray background.');

* }

*

* function draw() {

*   background(200);

*

*   // Enable orbiting with the mouse.

*   orbitControl();

*

*   // Draw the sphere.

*   sphere();

* }

* </code>

*

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white sphere on a gray background.');

* }

*

```
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the sphere.
*   // Set its radius to 30.
*   sphere(30);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white sphere on a gray background.');
* }
*
* function draw() {
*   background(200);
*
```

*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the sphere.
*   // Set its radius to 30.
*   // Set its detailX to 6.
*   sphere(30, 6);
* }
* </code>
*
*
*
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white sphere on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();

```
 *
 *   // Draw the sphere.
 *   // Set its radius to 30.
 *   // Set its detailX to 24.
 *   // Set its detailY to 4.
 *   sphere(30, 24, 4);
 * }
 * </code>
 * </div>
 */
p5.prototype.sphere = function(radius = 50, detailX = 24, detailY = 16) {
  this._assert3d('sphere');
  p5._validateParameters('sphere', arguments);

  this.ellipsoid(radius, radius, radius, detailX, detailY);

  return this;
};

/**
 * @private
 * Helper function for creating both cones and cylinders
 * Will only generate well-defined geometry when bottomRadius, height > 0
 * and topRadius >= 0
 * If topRadius == 0, topCap should be false
 */
```

```
const _truncatedCone = function(
  bottomRadius,
  topRadius,
  height,
  detailX,
  detailY,
  bottomCap,
  topCap
) {
  bottomRadius = bottomRadius <= 0 ? 1 : bottomRadius;
  topRadius = topRadius < 0 ? 0 : topRadius;
  height = height <= 0 ? bottomRadius : height;
  detailX = detailX < 3 ? 3 : detailX;
  detailY = detailY < 1 ? 1 : detailY;
  bottomCap = bottomCap === undefined ? true : bottomCap;
  topCap = topCap === undefined ? topRadius !== 0 : topCap;
  const start = bottomCap ? -2 : 0;
  const end = detailY + (topCap ? 2 : 0);
  //ensure constant slant for interior vertex normals
  const slant = Math.atan2(bottomRadius - topRadius, height);
  const sinSlant = Math.sin(slant);
  const cosSlant = Math.cos(slant);
  let yy, ii, jj;
  for (yy = start; yy <= end; ++yy) {
    let v = yy / detailY;
    let y = height * v;
```

```
let ringRadius;
if (yy < 0) {
  //for the bottomCap edge
  y = 0;
  v = 0;
  ringRadius = bottomRadius;
} else if (yy > detailY) {
  //for the topCap edge
  y = height;
  v = 1;
  ringRadius = topRadius;
} else {
  //for the middle
  ringRadius = bottomRadius + (topRadius - bottomRadius) * v;
}
if (yy === -2 || yy === detailY + 2) {
  //center of bottom or top caps
  ringRadius = 0;
}

y -= height / 2; //shift coordiate origin to the center of object
for (ii = 0; ii < detailX; ++ii) {
  const u = ii / (detailX - 1);
  const ur = 2 * Math.PI * u;
  const sur = Math.sin(ur);
  const cur = Math.cos(ur);
```

```
    //VERTICES
    this.vertices.push(new p5.Vector(sur * ringRadius, y, cur * ringRadius));

    //VERTEX NORMALS
    let vertexNormal;
    if (yy < 0) {
      vertexNormal = new p5.Vector(0, -1, 0);
    } else if (yy > detailY && topRadius) {
      vertexNormal = new p5.Vector(0, 1, 0);
    } else {
      vertexNormal = new p5.Vector(sur * cosSlant, sinSlant, cur * cosSlant);
    }
    this.vertexNormals.push(vertexNormal);
    //UVs
    this.uvs.push(u, v);
  }
}

let startIndex = 0;
if (bottomCap) {
  for (jj = 0; jj < detailX; ++jj) {
    const nextjj = (jj + 1) % detailX;
    this.faces.push([
      startIndex + jj,
      startIndex + detailX + nextjj,
```

```
      startIndex + detailX + jj
    ]);
  }
  startIndex += detailX * 2;
}
for (yy = 0; yy < detailY; ++yy) {
  for (ii = 0; ii < detailX; ++ii) {
    const nextii = (ii + 1) % detailX;
    this.faces.push([
      startIndex + ii,
      startIndex + nextii,
      startIndex + detailX + nextii
    ]);
    this.faces.push([
      startIndex + ii,
      startIndex + detailX + nextii,
      startIndex + detailX + ii
    ]);
  }
  startIndex += detailX;
}
if (topCap) {
  startIndex += detailX;
  for (ii = 0; ii < detailX; ++ii) {
    this.faces.push([
      startIndex + ii,
```

```
      startIndex + (ii + 1) % detailX,
      startIndex + detailX
    ]);
  }
 }
};
```

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

1. **Uninitialized or Null References:** The `radius`, `detailX`, and `detailY` parameters in the `sphere()` function can be undefined or uninitialized, leading to default values. However, there is no validation ensuring they are positive numbers, which could cause visual issues during rendering.

Category B: Data Handling Errors

2. **Handling of Detail Parameters (detailX, detailY):** While defaults are provided for `detailX` and `detailY`, there's no validation of their values. If these parameters are passed as invalid numbers (like negative integers), it could still lead to problematic rendering.

Category C: Computation Errors

3. **Geometry Buffer Creation:** In the `_truncatedCone()` function, if the calculation of vertex positions or the creation of geometry buffers does not account for edge cases (like degenerate shapes), it could lead to excessive memory usage or incorrect shapes rendered.

4. Edge Handling for Stroke: There is a condition that checks if strokes should be drawn for shapes with detail values above a certain threshold (like 4). If this condition isn't correctly enforced, it may lead to visual artefacts or improper rendering when lower detail levels are used.

Category D: Geometry Construction and Rendering

5. **Rendering Condition Errors:** The function relies on WebGL mode being active; if it's not properly checked or enforced, it can cause runtime errors during rendering.

```
 * Draws a cylinder.
 *
 * A cylinder is a 3D shape with triangular faces that connect a flat bottom
```

* to a flat top. Cylinders with few faces look like boxes. Cylinders with

* many faces have smooth surfaces.

*

* The first parameter, `radius`, is optional. If a `Number` is passed, as in

* `cylinder(20)`, it sets the radius of the cylinder’s base. By default,

* `radius` is 50.

*

* The second parameter, `height`, is also optional. If a `Number` is passed,

* as in `cylinder(20, 30)`, it sets the cylinder’s height. By default,

* `height` is set to the cylinder’s `radius`.

*

* The third parameter, `detailX`, is also optional. If a `Number` is passed,

* as in `cylinder(20, 30, 5)`, it sets the number of edges used to form the

* cylinder's top and bottom. Using more edges makes the top and bottom look

* more like circles. By default, `detailX` is 24.

*

* The fourth parameter, `detailY`, is also optional. If a `Number` is passed,

* as in `cylinder(20, 30, 5, 2)`, it sets the number of triangle subdivisions

* to use along the y-axis, between cylinder's the top and bottom. All 3D

* shapes are made by connecting triangles to form their surfaces. By default,

* `detailY` is 1.

*

* The fifth parameter, `bottomCap`, is also optional. If a `false` is passed,

* as in `cylinder(20, 30, 5, 2, false)` the cylinder’s bottom won’t be drawn.

* By default, `bottomCap` is `true`.

*

* The sixth parameter, `topCap`, is also optional. If a `false` is passed, as

* in `cylinder(20, 30, 5, 2, false, false)` the cylinder's top won't be

* drawn. By default, `topCap` is `true`.

*

* Note: `cylinder()` can only be used in WebGL mode.

*

* @method cylinder

* @param {Number} [radius]   radius of the cylinder. Defaults to 50.

* @param {Number} [height]   height of the cylinder. Defaults to the value of `radius`.

* @param {Integer} [detailX]   number of edges along the top and bottom. Defaults to 24.

* @param {Integer} [detailY]   number of triangle subdivisions along the y-axis. Defaults to 1.

* @param {Boolean} [bottomCap] whether to draw the cylinder's bottom. Defaults to `true`.

* @param {Boolean} [topCap]   whether to draw the cylinder's top. Defaults to `true`.

* @chainable

*

* @example

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white cylinder on a gray background.');

* }

*

```
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cylinder.
*   cylinder();
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cylinder on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
```

```
*   orbitControl();
*
*   // Draw the cylinder.
*   // Set its radius and height to 30.
*   cylinder(30);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cylinder on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cylinder.
```

*   // Set its radius to 30 and height to 50.

*   cylinder(30, 50);

* }

* </code>

*

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white box on a gray background.');

* }

*

* function draw() {

*   background(200);

*

*   // Enable orbiting with the mouse.

*   orbitControl();

*

*   // Draw the cylinder.

*   // Set its radius to 30 and height to 50.

*   // Set its detailX to 5.

*   cylinder(30, 50, 5);

```
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cylinder on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cylinder.
*   // Set its radius to 30 and height to 50.
*   // Set its detailX to 24 and detailY to 2.
*   cylinder(30, 50, 24, 2);
* }
* </code>
```

```
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cylinder on a gray background. Its top is missing.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cylinder.
*   // Set its radius to 30 and height to 50.
*   // Set its detailX to 24 and detailY to 1.
*   // Don't draw its bottom.
*   cylinder(30, 50, 24, 1, false);
* }
* </code>
* </div>
```

```
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cylinder on a gray background. Its top and bottom are missing.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cylinder.
*   // Set its radius to 30 and height to 50.
*   // Set its detailX to 24 and detailY to 1.
*   // Don't draw its bottom or top.
*   cylinder(30, 50, 24, 1, false, false);
* }
* </code>
* </div>
*/
```

```
p5.prototype.cylinder = function(
  radius = 50,
  height = radius,
  detailX = 24,
  detailY = 1,
  bottomCap = true,
  topCap = true
) {
  this._assert3d('cylinder');
  p5._validateParameters('cylinder', arguments);

  const gId = `cylinder|${detailX}|${detailY}|${bottomCap}|${topCap}`;
  if (!this._renderer.geometryInHash(gId)) {
    const cylinderGeom = new p5.Geometry(detailX, detailY);
    _truncatedCone.call(
      cylinderGeom,
      1,
      1,
      1,
      detailX,
      detailY,
      bottomCap,
      topCap
    );
    // normals are computed in call to _truncatedCone
    if (detailX <= 24 && detailY <= 16) {
```

```
    cylinderGeom._makeTriangleEdges()._edgesToVertices();
  } else if (this._renderer._doStroke) {
    console.log(
      'Cannot draw stroke on cylinder objects with more' +
      ' than 24 detailX or 16 detailY'
    );
  }
  this._renderer.createBuffers(gId, cylinderGeom);
}


this._renderer.drawBuffersScaled(gId, radius, height, radius);


return this;
};
```

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

1. **Uninitialized or Null References:** The `radius`, `height`, `detailX`, and `detailY` parameters can be passed as undefined or invalid values. Although defaults are set, there's no validation to ensure they are positive numbers, which could lead to incorrect visual outputs.

Category B: Data Handling Errors

2. **Handling of Detail Parameters (detailX, detailY):** The function defaults are used for `detailX` and `detailY`, but there is no validation of these parameters. If invalid values (such as negative integers) are passed, it could result in improper geometry creation.

Category C: Computation Errors

3. **Geometry Buffer Creation:** The check using `gId` for geometry caching relies on correct implementation. If the geometry is not cached properly or if there are issues in the creation of geometry buffers (for example, with unexpected parameters), it may lead to inefficient memory usage or incorrect rendering.

4. Edge Handling for Stroke: The function includes conditions to handle drawing strokes based on `detailX` and `detailY`. If the limits are exceeded (more than 24 for `detailX` or 16 for `detailY`), it logs a warning but does not prevent the rendering, which could lead to visual inconsistencies.

Category D: Rendering Conditions 5. **WebGL Mode Requirement:**

- The function relies on being executed in WebGL mode. If this mode is not correctly set, it will lead to runtime errors during execution, preventing the cylinder from rendering.

2. Which Category of Inspection is More Useful?

Category A: Data Reference Errors Validating input parameters (ensuring they are positive numbers) is crucial to prevent rendering issues. Addressing these checks will enhance the robustness of the `cylinder()` function and ensure correct output.

3. Which Type of Error is Not Identifiable Using Program Inspection?

Rendering Issues or Visual Artifacts While program inspection can verify logical correctness, it cannot visually confirm how the cylinder appears in various scenarios. Visual testing is necessary to detect any rendering inconsistencies or artifacts that may occur during execution.

4. Is Program Inspection Worth Applying?

Yes. Program inspection is advantageous for identifying potential issues related to input validation, geometry caching, and rendering logic. It helps ensure better performance, memory management, and overall code quality, reducing the likelihood of visual errors in rendering the cylinder. Regular inspection and validation can preemptively address issues before they impact the user experience.

* Draws a cone.

*

* A cone is a 3D shape with triangular faces that connect a flat bottom to a

* single point. Cones with few faces look like pyramids. Cones with many

* faces have smooth surfaces.

*

* The first parameter, `radius`, is optional. If a `Number` is passed, as in

* `cone(20)`, it sets the radius of the cone’s base. By default, `radius` is

* 50.

*

* The second parameter, `height`, is also optional. If a `Number` is passed,

* as in `cone(20, 30)`, it sets the cone’s height. By default, `height` is

* set to the cone’s `radius`.

*

* The third parameter, `detailX`, is also optional. If a `Number` is passed,

* as in `cone(20, 30, 5)`, it sets the number of edges used to form the

* cone's base. Using more edges makes the base look more like a circle. By

* default, `detailX` is 24.

*

* The fourth parameter, `detailY`, is also optional. If a `Number` is passed,

* as in `cone(20, 30, 5, 7)`, it sets the number of triangle subdivisions to

* use along the y-axis connecting the base to the tip. All 3D shapes are made

* by connecting triangles to form their surfaces. By default, `detailY` is 1.

*

* The fifth parameter, `cap`, is also optional. If a `false` is passed, as

* in `cone(20, 30, 5, 7, false)` the cone’s base won’t be drawn. By default,

* `cap` is `true`.

*

* Note: `cone()` can only be used in WebGL mode.

*

* @method cone

```
* @param  {Number}  [radius]  radius of the cone's base. Defaults to 50.
* @param  {Number}  [height]  height of the cone. Defaults to the value of `radius`.
* @param  {Integer} [detailX] number of edges used to draw the base. Defaults to 24.
* @param  {Integer} [detailY] number of triangle subdivisions along the y-axis. Defaults to 1.
* @param  {Boolean} [cap]     whether to draw the cone's base.  Defaults to `true`.
* @chainable
*
* @example
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cone on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cone.
*   cone();
```

```
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cone on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cone.
*   // Set its radius and height to 30.
*   cone(30);
* }
* </code>
* </div>
```

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white cone on a gray background.');

* }

*

* function draw() {

*   background(200);

*

*   // Enable orbiting with the mouse.

*   orbitControl();

*

*   // Draw the cone.

*   // Set its radius to 30 and height to 50.

*   cone(30, 50);

* }

* </code>

*

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white cone on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cone.
*   // Set its radius to 30 and height to 50.
*   // Set its detailX to 5.
*   cone(30, 50, 5);
* }
* </code>
*
*
*
* <code>
* // Click and drag the mouse to view the scene from different angles.
*

```
* function setup() {
*   createCanvas(100, 100, WEBGL);
*
*   describe('A white pyramid on a gray background.');
* }
*
* function draw() {
*   background(200);
*
*   // Enable orbiting with the mouse.
*   orbitControl();
*
*   // Draw the cone.
*   // Set its radius to 30 and height to 50.
*   // Set its detailX to 5.
*   cone(30, 50, 5);
* }
* </code>
* </div>
*
* <div>
* <code>
* // Click and drag the mouse to view the scene from different angles.
*
* function setup() {
*   createCanvas(100, 100, WEBGL);
```

*

*   describe('A white cone on a gray background.');

* }

*

* function draw() {

*   background(200);

*

*   // Enable orbiting with the mouse.

*   orbitControl();

*

*   // Draw the cone.

*   // Set its radius to 30 and height to 50.

*   // Set its detailX to 24 and detailY to 2.

*   cone(30, 50, 24, 2);

* }

* </code>

*

*

*

* <code>

* // Click and drag the mouse to view the scene from different angles.

*

* function setup() {

*   createCanvas(100, 100, WEBGL);

*

*   describe('A white cone on a gray background. Its base is missing.');

```
 * }
 *
 * function draw() {
 *   background(200);
 *
 *   // Enable orbiting with the mouse.
 *   orbitControl();
 *
 *   // Draw the cone.
 *   // Set its radius to 30 and height to 50.
 *   // Set its detailX to 24 and detailY to 1.
 *   // Don't draw its base.
 *   cone(30, 50, 24, 1, false);
 * }
 * </code>
 * </div>
 */
p5.prototype.cone = function(
  radius = 50,
  height = radius,
  detailX = 24,
  detailY = 1,
  cap = true
) {
  this._assert3d('cone');
  p5._validateParameters('cone', arguments);
```

```
  const gId = `cone|${detailX}|${detailY}|${cap}`;
  if (!this._renderer.geometryInHash(gId)) {
    const coneGeom = new p5.Geometry(detailX, detailY);
    _truncatedCone.call(coneGeom, 1, 0, 1, detailX, detailY, cap, false);
    if (detailX <= 24 && detailY <= 16) {
      coneGeom._makeTriangleEdges()._edgesToVertices();
    } else if (this._renderer._doStroke) {
      console.log(
        'Cannot draw stroke on cone objects with more' +
        ' than 24 detailX or 16 detailY'
      );
    }
    this._renderer.createBuffers(gId, coneGeom);
  }

  this._renderer.drawBuffersScaled(gId, radius, height, radius);

  return this;
};
```

1. How many errors are there in the program? Mention the errors you have identified.

Category A: Data Reference Errors

1. **Uninitialized or Null References:** The parameters `radius`, `height`, `detailX`, and `detailY` can be passed as undefined or invalid values. While defaults are provided, there is no validation to ensure they are positive numbers, which could lead to rendering problems.

Category B: Data Handling Errors

2. **Handling of Detail Parameters (detailX, detailY):** There is no validation for `detailX` and `detailY`. If these parameters are set to invalid values (such as negative integers), it may result in improper geometry creation, leading to visual inconsistencies.

Category C: Computation Errors

3. **Geometry Buffer Creation:** The geometry caching mechanism using `gId` relies on correct implementation. If the geometry is not cached correctly or if there are issues in creating geometry buffers, it may result in inefficient memory usage or incorrect rendering.

4. Edge Handling for Stroke: The function checks to handle drawing strokes based on `detailX` and `detailY`. If the limits are exceeded (more than 24 for `detailX` or 16 for `detailY`), it logs a warning but does not prevent the drawing operation, which could lead to visual artifacts.

Category D: Rendering Conditions

5. **WebGL Mode Requirement:** The `cone()` function must be executed in WebGL mode. If the WebGL context is not established, it will lead to runtime errors during execution, preventing the cone from rendering properly.

2. Which Category of Inspection is More Useful?

Category A: Data Reference Errors Validating input parameters (ensuring they are positive numbers) is essential to prevent rendering issues. Addressing these checks will enhance the robustness of the `cone()` function and ensure the correct output.

---

3. Which Type of Error is Not Identifiable Using Program Inspection?

Rendering Issues or Visual Artifacts While program inspection can verify the logical correctness of the code, it cannot visually confirm how the cone appears under different rendering scenarios. Visual testing is necessary to detect any rendering inconsistencies or artifacts.

---

4. Is Program Inspection Worth Applying?

Yes. Program inspection is beneficial for identifying potential issues related to input validation, geometry caching, and rendering logic. It helps ensure better performance, memory management, and overall code quality, thereby reducing the likelihood of visual errors during the rendering of the cone. Regular inspections and validations can preemptively address issues, enhancing user experience and reliability.