02561 Computer Graphics

3D models and asynchronous data loading

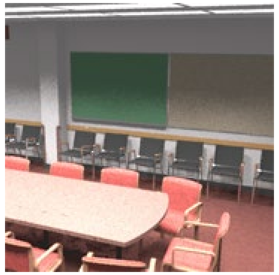
Jeppe Revall Frisvad

October 2024

Acquiring 3D models

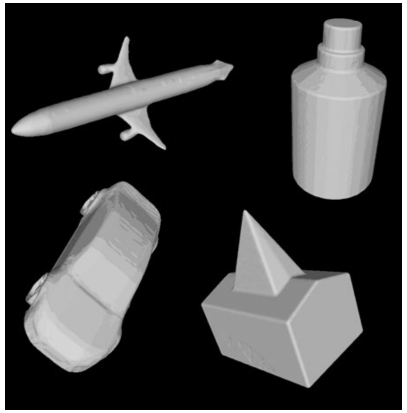
- Stanford 3D scanning repository
  https://graphics.stanford.edu/data/3Dscanrep/

- McGuire computer graphics archive
  https://casual-effects.com/data/

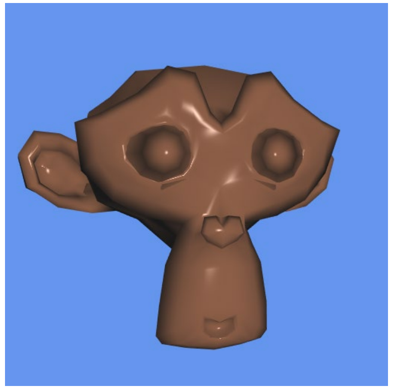


- Thingiverse
  https://www.thingiverse.com/

- ShapeNet
  https://www.shapenet.org/

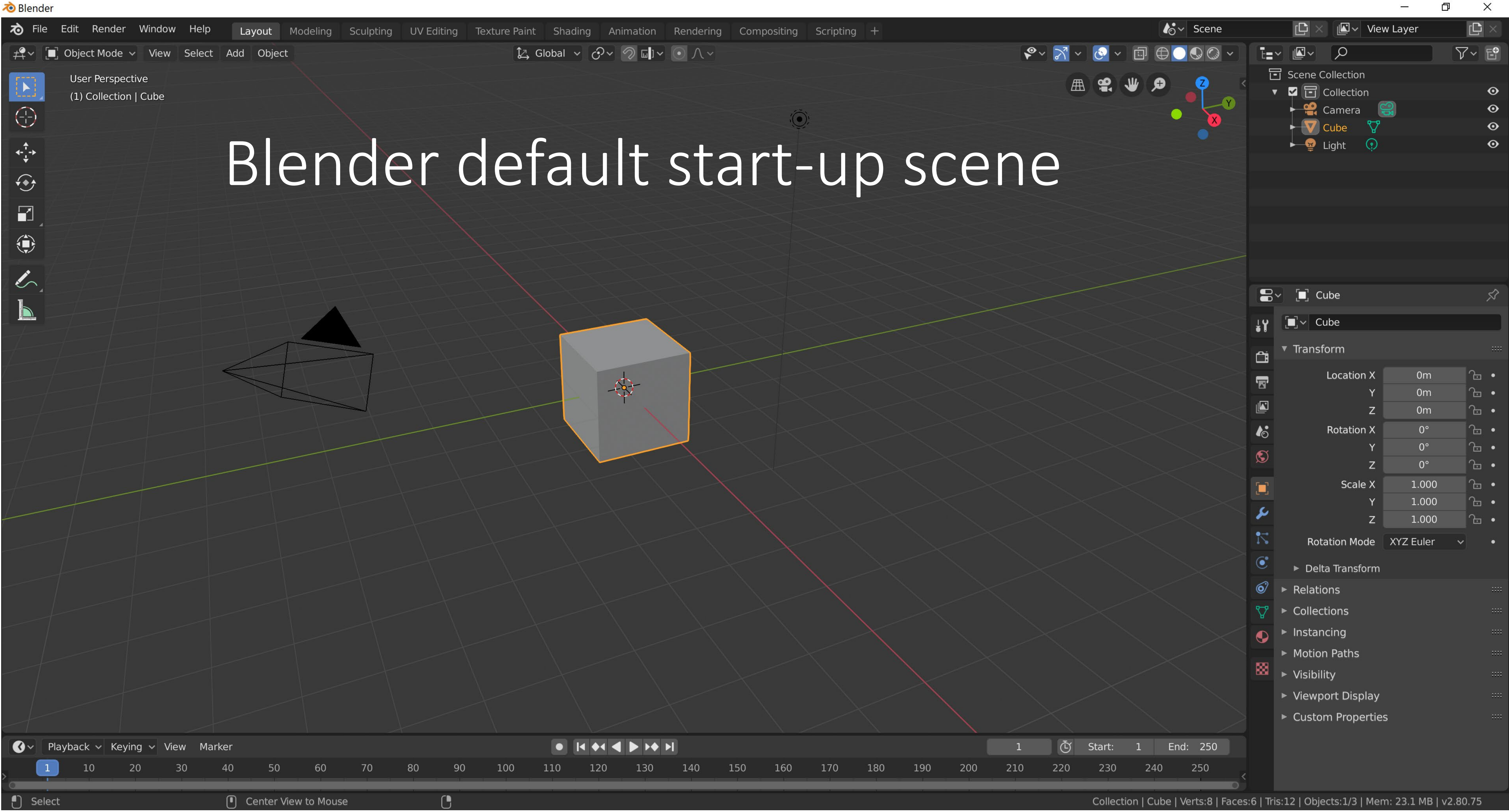


- Modeling tools:
  - Maya https://www.autodesk.com/products/maya/
  - Blender https://www.blender.org/
  - …



Export from Blender

1. Click the splash screen and right click and Delete the default cube.
2. Create a model (start from the Add → Mesh menu, for example).
3. Right click and select Shade Smooth to get interpolated vertex normals.
4. Export mesh: File → Export → Wavefront (.obj)
5. Select export options (deselect Include UVs, for example).
6. Choose folder and .obj file name and press Export OBJ.



Blender default start-up scene

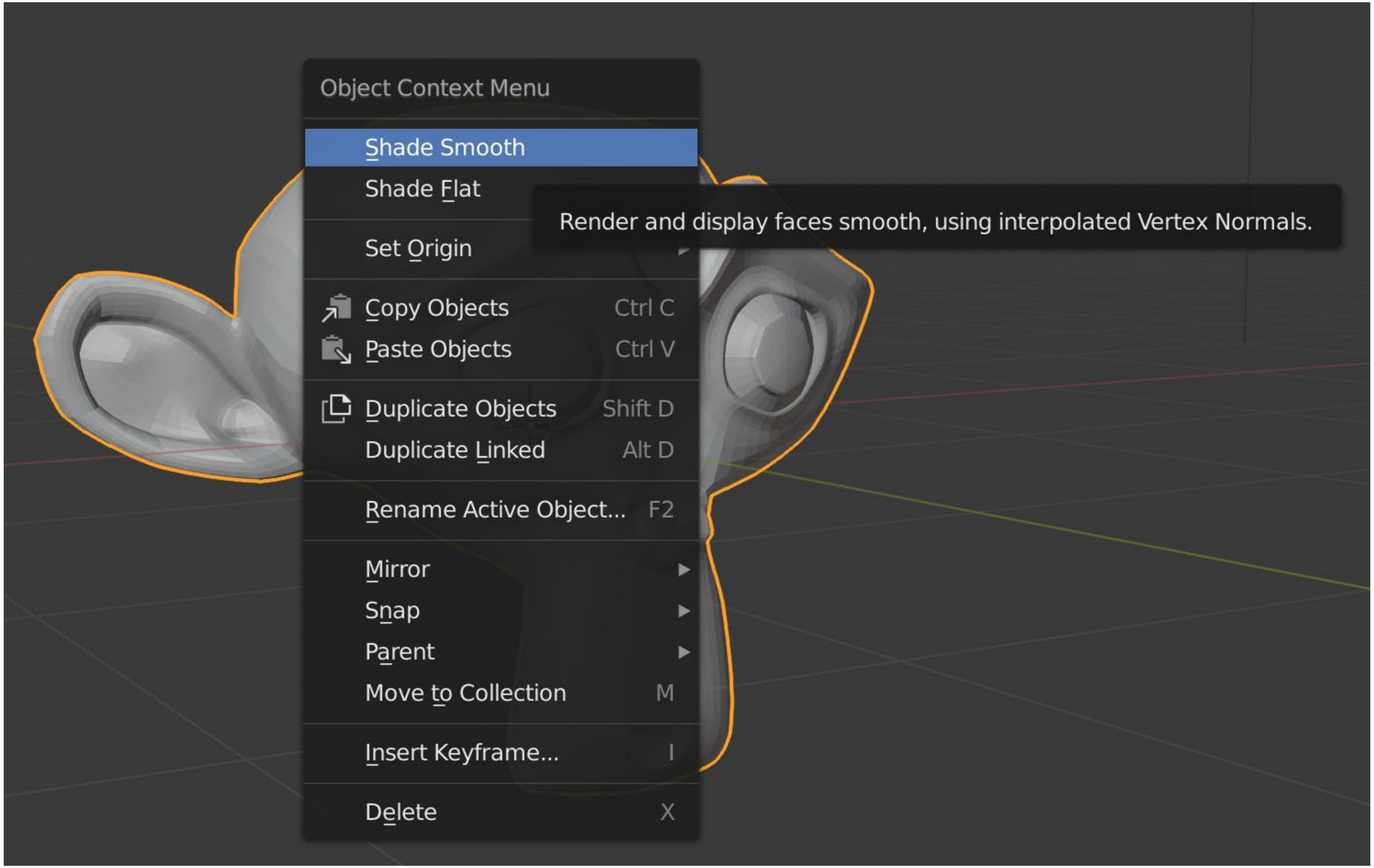
User Perspective
(1) Collection | Cube

Location X 0m
Y 0m
Z 0m
Rotation X 0°
Y 0°
Z 0°
Scale X 1.000
Y 1.000
Z 1.000
Rotation Mode XYZ Euler

Start: 1 End: 250

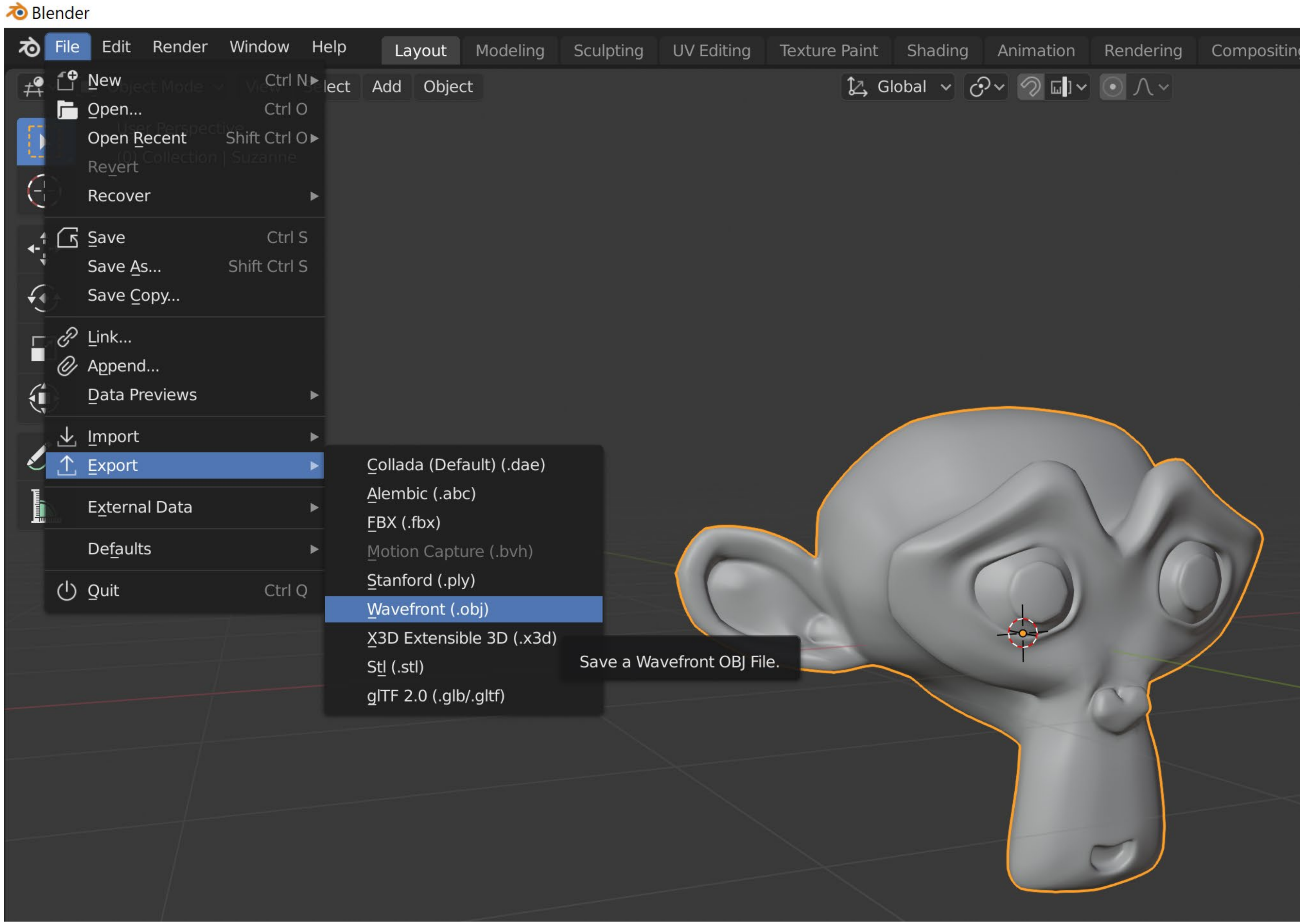
Collection | Cube | Verts:8 | Faces:6 | Tris:12 | Objects:1/3 | Mem: 23.1 MB | v2.80.75

Right click to get the Object Context Menu

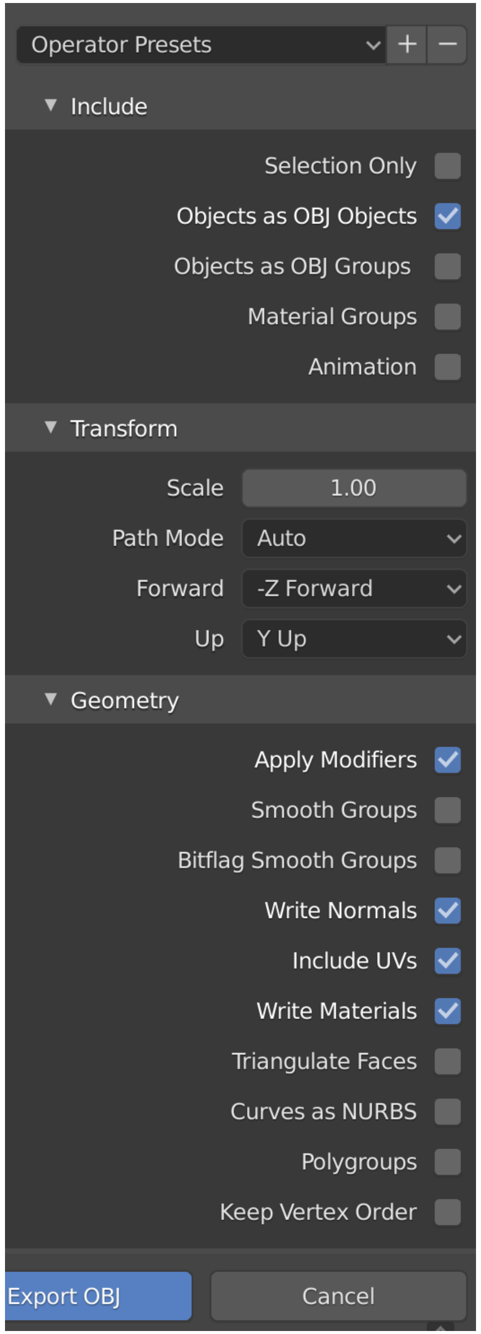


Render and display faces smooth, using interpolated Vertex Normals.

Blender export to Wavefront OBJ



Save a Wavefront OBJ File.



Include

- Selection Only □
- Objects as OBJ Objects ☑
- Objects as OBJ Groups □
- Material Groups □
- Animation □

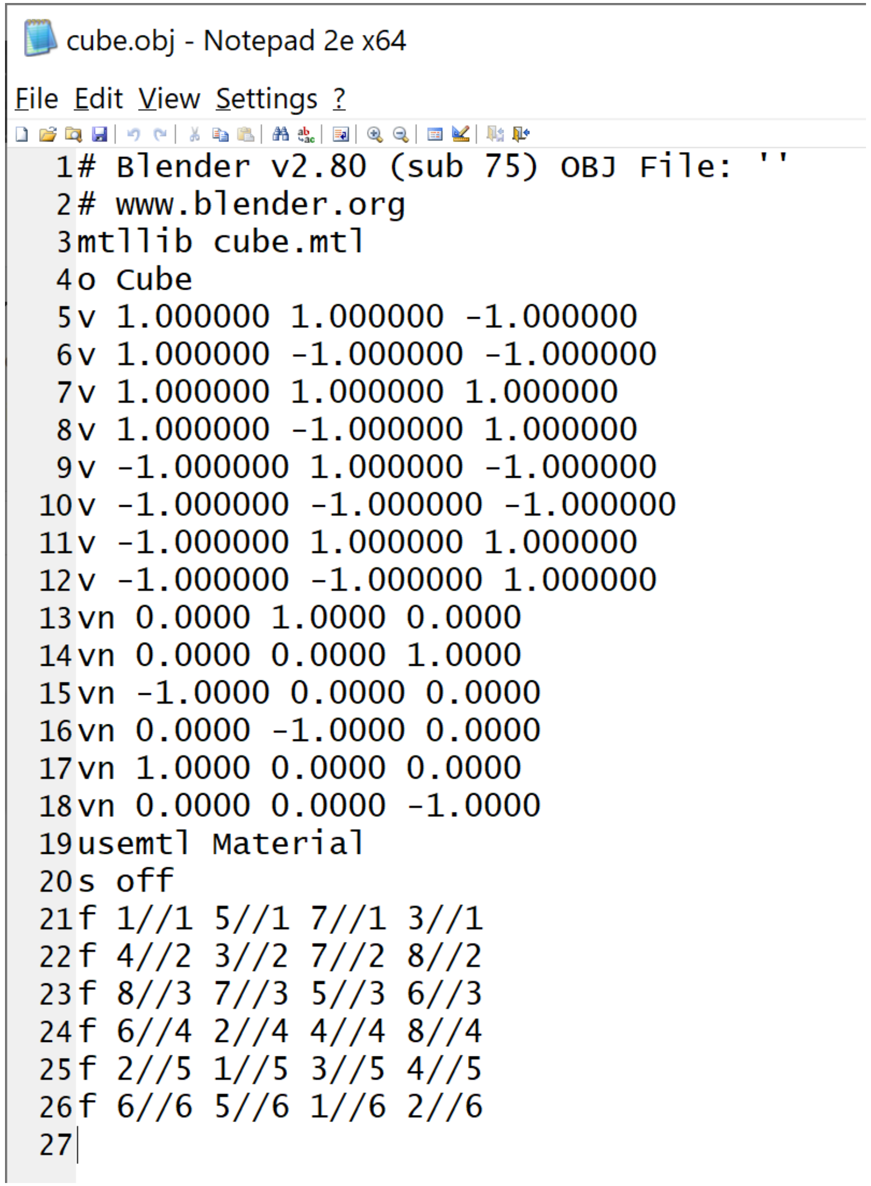
Transform

- Scale: 1.00
- Path Mode: Auto
- Forward: -Z Forward
- Up: Y Up

Geometry

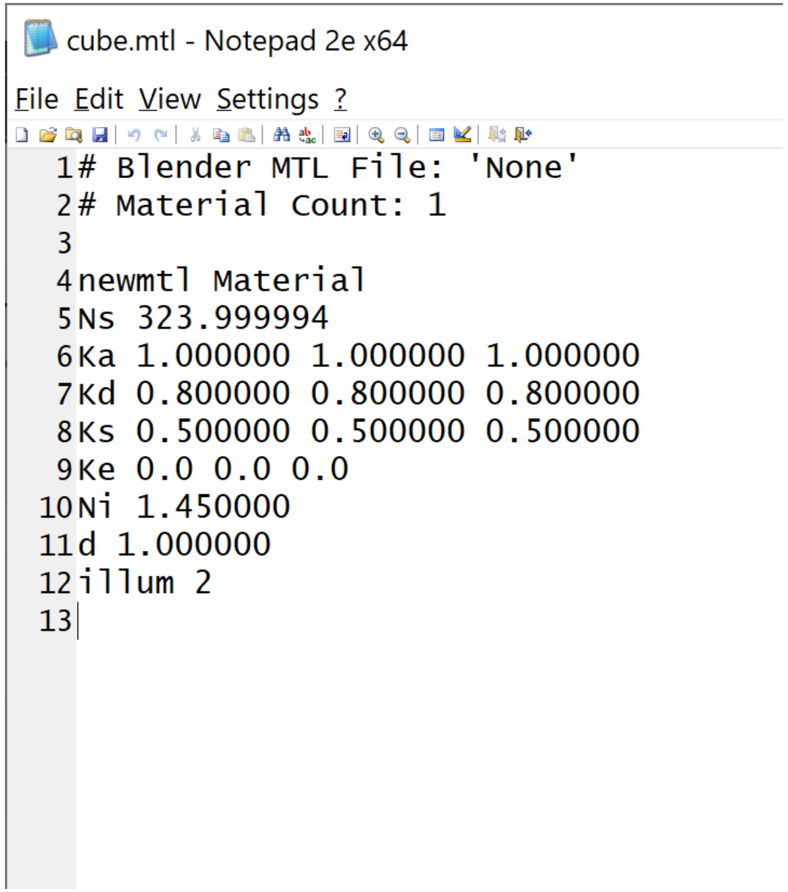
- Apply Modifiers ☑
- Smooth Groups □
- Bitflag Smooth Groups □
- Write Normals ☑
- Include UVs ☑
- Write Materials ☑
- Triangulate Faces □
- Curves as NURBS □
- Polygroups □
- Keep Vertex Order □

Wavefront OBJ (.obj and .mtl files)



```
# Blender v2.80 (sub 75) OBJ File: ''
# www.blender.org
mtllib cube.mtl
o Cube
v 1.000000 1.000000 -1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 1.000000 1.000000
v -1.000000 -1.000000 1.000000
vn 0.0000 1.0000 0.0000
vn 0.0000 0.0000 1.0000
vn -1.0000 0.0000 0.0000
vn 0.0000 -1.0000 0.0000
vn 1.0000 0.0000 0.0000
vn 0.0000 0.0000 -1.0000
usemtl Material
s off
f 1//1 5//1 7//1 3//1
f 4//2 3//2 7//2 8//2
f 8//3 7//3 5//3 6//3
f 6//4 2//4 4//4 8//4
f 2//5 1//5 3//5 4//5
f 6//6 5//6 1//6 2//6

```

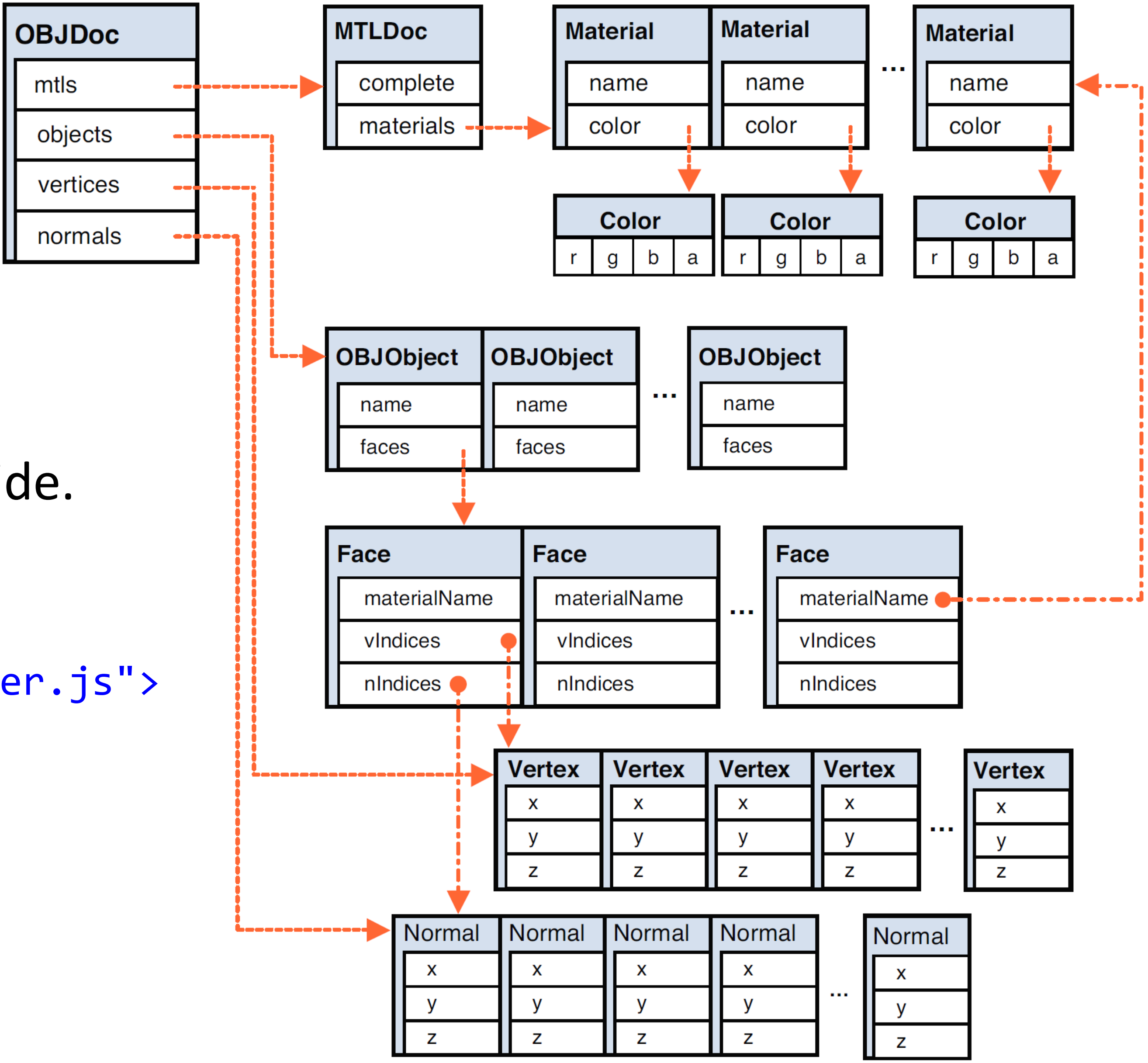


```
# Blender MTL File: 'None'
# Material Count: 1

newmtl Material
Ns 323.999994
Ka 1.000000 1.000000 1.000000
Kd 0.800000 0.800000 0.800000
Ks 0.500000 0.500000 0.500000
Ke 0.0 0.0 0.0
Ni 1.450000
d 1.000000
illum 2

```

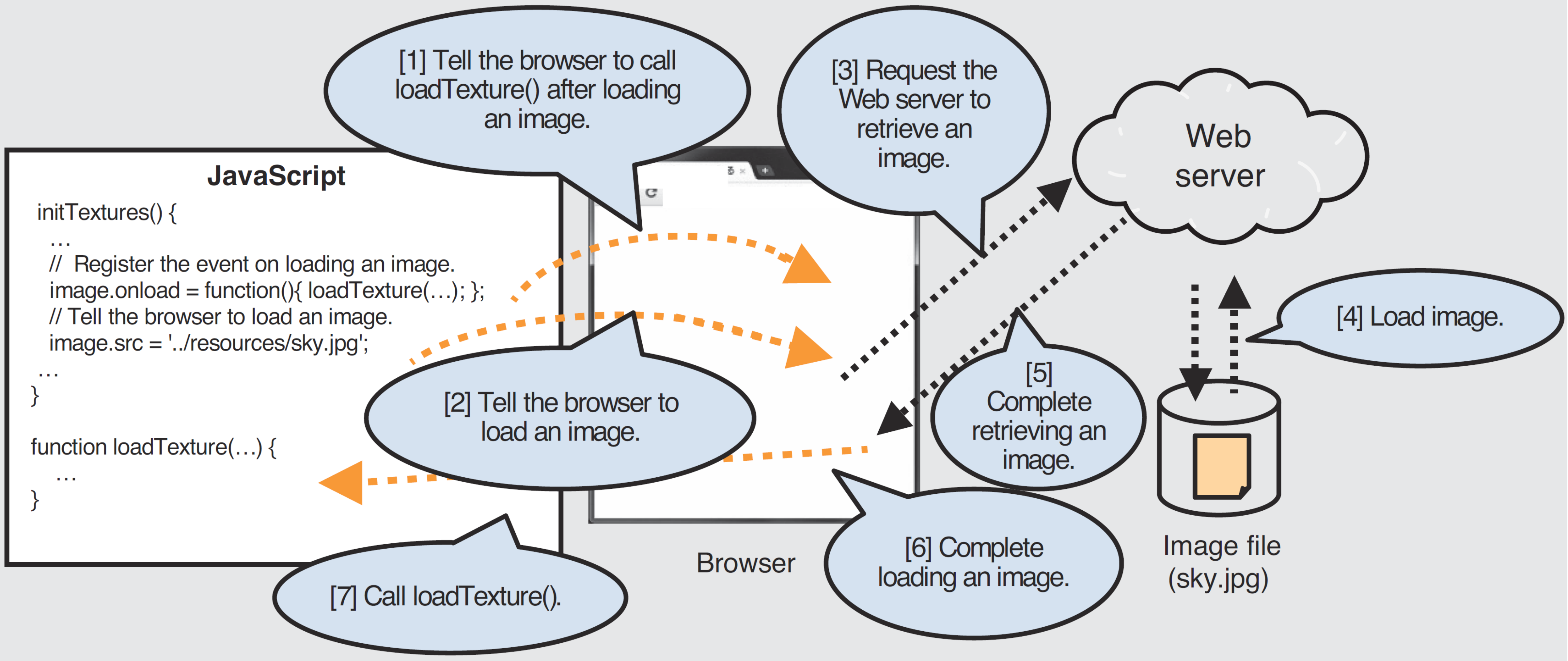
Parsing Wavefront OBJ files



Implemented in **OBJParser.js**
from WebGL Programming Guide.

```
<script type="text/javascript"
        src="../common/OBJParser.js">
</script>
```

Asynchronous data loading



JavaScript

```
initTextures() {
  …
  //  Register the event on loading an image.
  image.onload = function(){ loadTexture(…); };
  // Tell the browser to load an image.
  image.src = '../resources/sky.jpg';
 …
}

function loadTexture(…) {
    …
}
```

[1] Tell the browser to call loadTexture() after loading an image.
[2] Tell the browser to load an image.
[3] Request the Web server to retrieve an image.
Web server
[4] Load image.
[5] Complete retrieving an image.
[6] Complete loading an image.
Image file (sky.jpg)
Browser
[7] Call loadTexture().

When loading an OBJ file:

`onReadOBJFile` corresponds to `image.onload`

`image.src` becomes an http request to open a file

`onReadComplete` corresponds to `loadTexture`

Loading an OBJ file into WebGL

- Due to asynchronous data loading, we load an OBJ file using `async` and `await`:

```
window.onload = async function init()
{
  const obj_filename = "suzanne_smooth.obj";
  const drawingInfo = await readOBJFile(obj_filename, 1.0, true);
  ⋮
}
```

- The drawing info contains an indexed face set with indices as well as vertex positions, normals, and colors. The data for the buffers are in
  - `drawingInfo.vertices, drawingInfo.normals, drawingInfo.colors`
  - `drawingInfo.indices`

- Positions, normals, and colors are 4-vectors.
- Indices are 32-bit unsigned integers (extension needed).

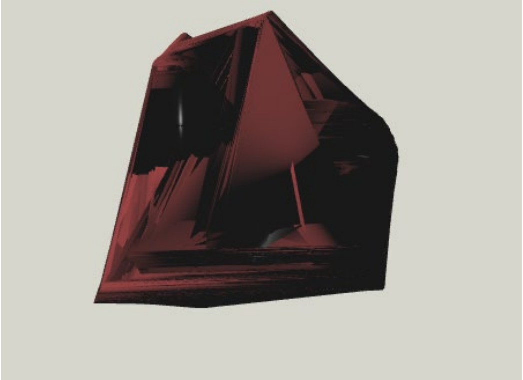
File access from files

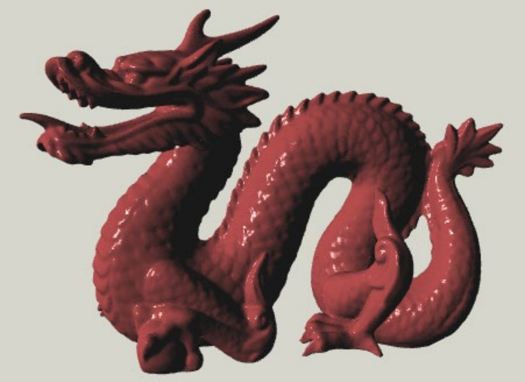
- Local file access is restricted to maintain browser security.
- To work locally, use Python to set up a local server:
    - Open a command prompt
    - Go to folder with your solution files and library files (possibly in subfolders).
    - Write the command (might vary slightly on different systems):

```
python –m http.server
```

- Alternative: Upload your webpage to a server and run your program by visiting the webpage (then you are no longer working locally).
- You have a student webspace https://www.student.dtu.dk/~username/ http://gbar.dtu.dk/faq/50-homepage
- Upload files using SCP or SFTP (we recommend WinSCP for Windows).

Extension: high poly count objects





- Let us try to render a high poly object.
- The Stanford dragon is ~560k vertices, ~1100k triangles.
- When trying, it seems to be inside-out.
- We need an extension: `OES_element_index_uint`
  https://www.khronos.org/registry/webgl/extensions/

```
var ext = gl.getExtension('OES_element_index_uint');
if (!ext) {
  console.log('Warning: Unable to use an extension');
}
```

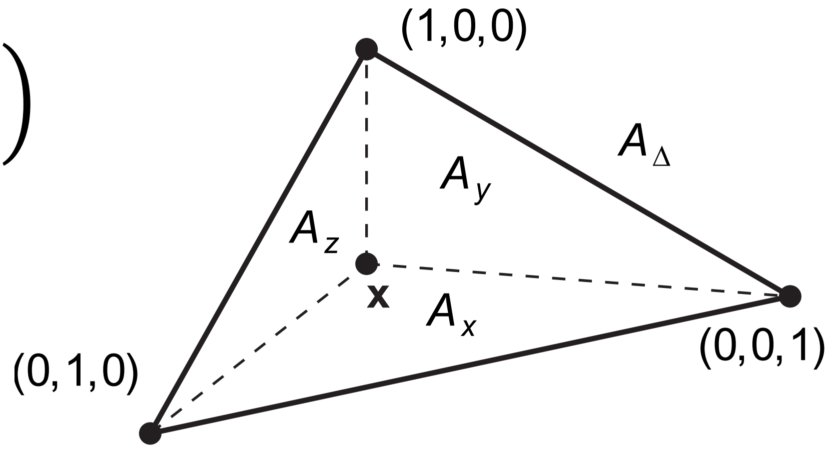
- This enables use of `gl.UNSIGNED_INT` (32 bits) instead of `gl.UNSIGNED_SHORT` (16 bits) when rendering an indexed face set. In OBJParser.js:

```
var indices = new Uint32Array(numIndices);
```

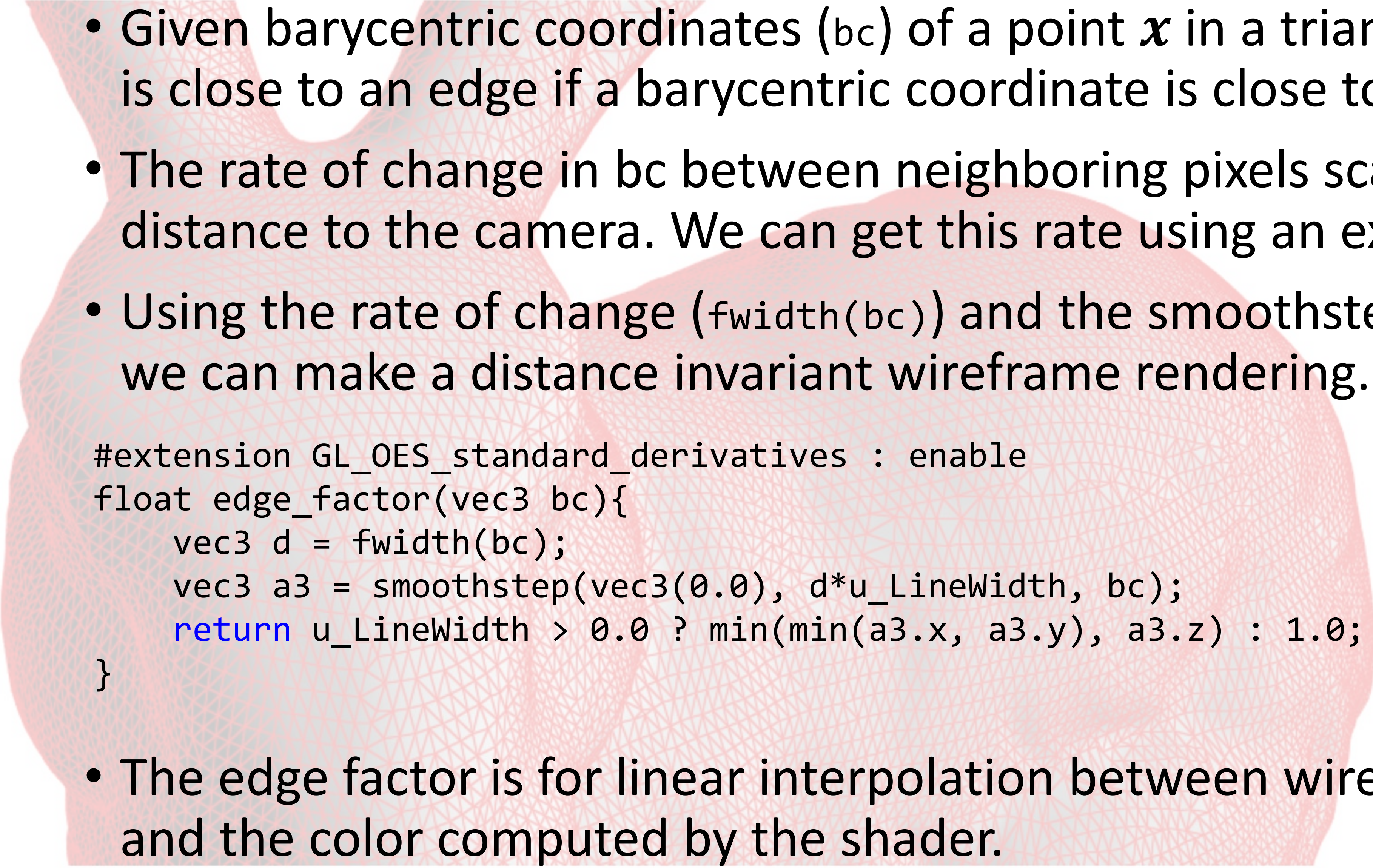
- Then draw using: `gl.drawElements(gl.TRIANGLES, drawingInfo.indices.length, gl.UNSIGNED_INT, 0);`

Extension: adding an attribute

- Sometimes it is useful to be able to add another vertex attribute.
- This can be done in the parser functions `getDrawingInfo` and `DrawingInfo`.
- Example 1:
  - For triangle vertices $\boldsymbol{v}_0, \boldsymbol{v}_1, \boldsymbol{v}_2$, we could add the corresponding attributes $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$.
  - Assigning this attribute to a varying variable in the vertex shader, it would become barycentric coordinates in the fragment shader.
- Barycentric coordinates: $(\alpha, \beta, \gamma) = \left(\frac{A_x}{A_\Delta}, \frac{A_y}{A_\Delta}, \frac{A_z}{A_\Delta}\right)$
- $\boldsymbol{x} = \alpha \boldsymbol{v}_0 + \beta \boldsymbol{v}_1 + \gamma \boldsymbol{v}_2$
- These are useful for wireframe rendering among many other things.
- We could also add the Bézier control point attribute from Week 3.

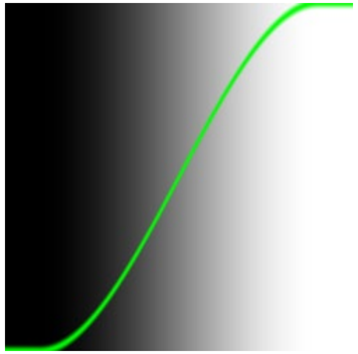


Wireframe rendering



- Given barycentric coordinates (`bc`) of a point x in a triangle, the point is close to an edge if a barycentric coordinate is close to 0.
- The rate of change in bc between neighboring pixels scales with the distance to the camera. We can get this rate using an extension.
- Using the rate of change (`fwidth(bc)`) and the smoothstep function, we can make a distance invariant wireframe rendering.

```
#extension GL_OES_standard_derivatives : enable
float edge_factor(vec3 bc){
    vec3 d = fwidth(bc);
    vec3 a3 = smoothstep(vec3(0.0), d*u_LineWidth, bc);
    return u_LineWidth > 0.0 ? min(min(a3.x, a3.y), a3.z) : 1.0;
}
```



smoothstep

- The edge factor is for linear interpolation between wireframe color and the color computed by the shader.