# **Assimp**

In all the scenes so far we've been extensively playing with our little container friend, but over time, even our best friends can get a little boring. In bigger graphics applications, there are usually lots of complicated and interesting models that are much prettier to look at than a static container. However, unlike the container object, we can't really manually define all the vertices, normals, and texture coordinates of complicated shapes like houses, vehicles, or human-like characters. What we want instead, is to *import* these models into the application; models that were carefully designed by 3D artists in tools like [Blender](http://www.blender.org/), [3DS Max](http://www.autodesk.nl/products/3ds-max/overview) or [Maya](http://www.autodesk.com/products/autodesk-maya/overview).

These so-called 3D modeling tools allow artists to create complicated shapes and apply textures to them via a process called uv-mapping. The tools then automatically generate all the vertex coordinates, vertex normals, and texture coordinates while exporting them to a model file format we can use. This way, artists have an extensive toolkit to create high quality models without having to care too much about the technical details. All the technical aspects are hidden in the exported model file. We, as graphics programmers, **do** have to care about these technical details though.

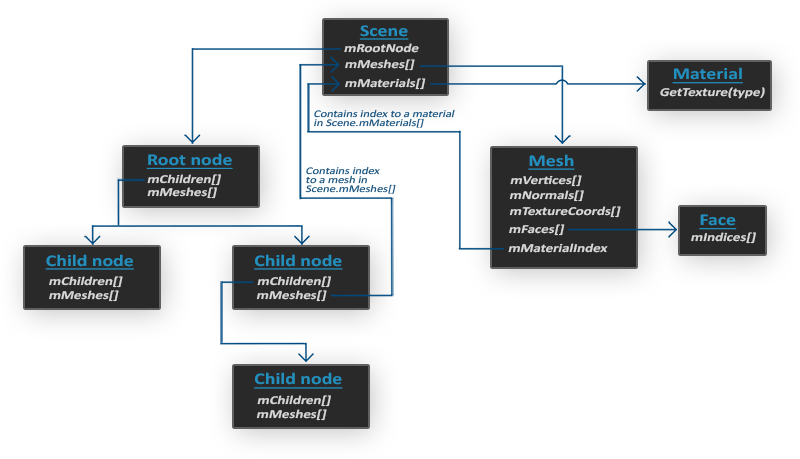
It is our job to parse these exported model files and extract all the relevant information so we can store them in a format that OpenGL understands. A common issue is that there are dozens of different file formats where each exports the model data in its own unique way. Model formats like the [Wavefront .obj](http://en.wikipedia.org/wiki/Wavefront_.obj_file) only contains model data with minor material information like model colors and diffuse/specular maps, while model formats like the XML-based [Collada file format](http://en.wikipedia.org/wiki/COLLADA) are extremely extensive and contain models, lights, many types of materials, animation data, cameras, complete scene information, and much more. The wavefront object format is generally considered to be an easy-to-parse model format. It is recommended to visit the Wavefront's wiki page at least once to see how such a file format's data is structured. This should give you a basic perception of how model file formats are generally structured.

All by all, there are many different file formats where a common general structure between them usually does not exist. So if we want to import a model from these file formats, we'd have to write an importer ourselves for each of the file formats we want to import. Luckily for us, there just happens to be a library for this.

## **A model loading library**

A very popular model importing library out there is called [Assimp](http://assimp.org/) that stands for *Open Asset Import Library* (we will be using a wrapper library called [AssimpNet](https://github.com/assimp/assimp-net)). Assimp is able to import dozens of different model file formats (and export to some as well) by loading all the model's data into Assimp's generalized data structures. As soon as Assimp has loaded the model, we can retrieve all the data we need from Assimp's data structures. Because the data structure of Assimp stays the same, regardless of the type of file format we imported, it abstracts us from all the different file formats out there.

When importing a model via Assimp it loads the entire model into a *scene* object that contains all the data of the imported model/scene. Assimp then has a collection of nodes where each node contains indices to data stored in the scene object where each node can have any number of children. A (simplistic) model of Assimp's structure is shown below:



* All the data of the scene/model is contained in the Scene object like all the materials and the meshes. It also contains a reference to the root node of the scene.
* The Root node of the scene may contain children nodes (like all other nodes) and could have a set of indices that point to mesh data in the scene object's mMeshes array. The scene's mMeshes (just Meshes for AssimpNet) array contains the actual Mesh objects, the values in the mMeshes array of a node are only indices for the scene's meshes array.
* A Mesh object itself contains all the relevant data required for rendering, think of vertex positions, normal vectors, texture coordinates, faces, and the material of the object.
* A mesh contains several faces. A Face represents a render primitive of the object (triangles, squares, points). A face contains the indices of the vertices that form a primitive. Because the vertices and the indices are separated, this makes it easy for us to render via an index buffer (see [Hello Triangle](https://learnopengl.com/Getting-started/Hello-Triangle)).
* Finally a mesh also links to a Material object that hosts several functions to retrieve the material properties of an object. Think of colors and/or texture maps (like diffuse and specular maps).

What we want to do is: first load an object into a Scene object, recursively retrieve the corresponding Mesh objects from each of the nodes (we recursively search each node's children), and process each Mesh object to retrieve the vertex data, indices, and its material properties. The result is then a collection of mesh data that we want to contain in a single Model object.

**Mesh**

When modeling objects in modeling toolkits, artists generally do not create an entire model out of a single shape. Usually, each model has several sub-models/shapes that it consists of. Each of those single shapes is called a mesh. Think of a human-like character: artists usually model the head, limbs, clothes, and weapons all as separate components, and the combined result of all these meshes represents the final model. A single mesh is the minimal representation of what we need to draw an object in OpenGL (vertex data, indices, and material properties). A model (usually) consists of several meshes.

In the [next](https://learnopengl.com/Model-Loading/Mesh) chapters we'll create our own Model and Mesh class that load and store imported models using the structure we've just described. If we then want to draw a model, we do not render the model as a whole, but we render all of the individual meshes that the model is composed of. However, before we can start importing models, we first need to actually include Assimp in our project.

## **Adding AssimpNet via NuGet**

## You can easily add AssimpNet to your project using the NuGet package manager. Search for AssimpNet by Nicholas Woodfield.

# **Mesh**

## With Assimp we can load many different models into the application, but once loaded they're all stored in Assimp's data structures. What we eventually want is to transform that data to a format that OpenGL understands so that we can render the objects. We learned from the previous chapter that a mesh represents a single drawable entity, so let's start by defining a mesh class of our own.

## Let's review a bit of what we've learned so far to think about what a mesh should minimally have as its data. A mesh should at least need a set of vertices, where each vertex contains a position vector, a normal vector, and a texture coordinate vector. A mesh should also contain indices for indexed drawing, and may contain material data in the form of textures (diffuse/specular maps).

## Now that we set the minimal requirements for a mesh class we can define a vertex in OpenGL:

## **public struct Vertex**

## **{**

## **public Vector3 Position;**

## **public Vector3 Normal;**

## **public Vector2 TexCoords;**

## **}**

## Knowing the actual representation of a vertex and a texture we can start defining the structure of the mesh class:

## **public class Mesh**

## **{**

## **public readonly int VAO;**

## **public readonly int indicesCount;**

## 

## **public Mesh(Span<Vertex> vertices, Span<int> indices); public void Draw();**

## **}**

## As you can see, the class isn't too complicated. In the constructor we give the mesh all the necessary data and initialize the buffers, then draw the mesh via the Draw function.

## **Constructor**

## Since our VAO contains a VBO and EBO, all of the vertices and indices will be stored within those buffers respectively. All we need to do is set up the appropriate buffers and specify the vertex shader layout via vertex attribute pointers. By now you should have no trouble with these concepts.

## **private void SetupMesh()**

## **{**

## **VAO = GL.GenVertexArray();**

## **int VBO = GL.GenBuffer();**

## **int EBO = GL.GenBuffer();**

## 

## **GL.BindVertexArray(VAO);**

## **GL.BindBuffer(BufferTarget.ArrayBuffer, VBO);**

## **GL.BufferData(BufferTarget.ArrayBuffer, vertices.Length \* Unsafe.SizeOf<Vertex>(), ref MemoryMarshal.GetReference(vertices), BufferUsageHint.StaticDraw);**

## 

## **GL.BindBuffer(BufferTarget.ElementArrayBuffer, EBO);**

## **GL.BufferData(BufferTarget.ElementArrayBuffer, indices.Length \* sizeof(int), ref MemoryMarshal.GetReference(indices), BufferUsageHint.StaticDraw);**

## 

## **// Vertex positions**

## **GL.EnableVertexAttribArray(0);**

## **GL.VertexAttribPointer(0, 3, VertexAttribPointerType.Float, false, Unsafe.SizeOf<Vertex>(), Marshal.OffsetOf<Vertex>(nameof(Vertex.Position)));**

## **// Vertex normals**

## **GL.EnableVertexAttribArray(1);**

## **GL.VertexAttribPointer(1, 3, VertexAttribPointerType.Float, false, Unsafe.SizeOf<Vertex>(), Marshal.OffsetOf<Vertex>(nameof(Vertex.Normal)));**

## **// Vertex texture coords**

## **GL.EnableVertexAttribArray(2);**

## **GL.VertexAttribPointer(2, 2, VertexAttribPointerType.Float, false, Unsafe.SizeOf<Vertex>(), Marshal.OffsetOf<Vertex>(nameof(Vertex.TexCoords)));**

## 

## **GL.BindVertexArray(0);**

## **}**

## Notice that we do several things to simplify our VAO:

## We use Unsafe.SizeOf() to get the size of a Vertex struct in memory.

## We use MemoryMarshal.GetReference() to pass the Span of vertices and the Span of indices to the OpenGL buffers.

* We use Marshal.OffsetOf() to get the offset in memory for each of the members of the Vertex Struct.

## **Rendering**

## The last function we need to define for the Mesh class to be complete is its Draw function. Before rendering the mesh, we will need to set which Shader we are using and bind the appropriate textures before calling GL.DrawElements. For simplicity, within the Draw function itself we will only bind the VAO, call GL.DrawElements, then unbind the VAO.

## **public void Draw()**

## **{ // Draw mesh**

## **GL.BindVertexArray(VAO);**

## **GL.DrawElements(PrimitiveType.Triangles, indices.Length, DrawElementsType.UnsignedInt, 0);**

## **GL.BindVertexArray(0);**

## **}**

## You can find the full source code of the Mesh class here.

## The Mesh class we just defined is an abstraction for many of the topics we've discussed in the early chapters. In the next chapter we'll create a model that acts as a container for several mesh objects and implements Assimp's loading interface.

# **Model**

## Now it is time to get our hands dirty with Assimp and start creating the actual loading and translation code. The goal of this chapter is to create another class that represents a model in its entirety, that is, a model that contains multiple meshes, possibly with multiple textures. A house that contains a wooden balcony, a tower, and perhaps a swimming pool, could still be loaded as a single model. We'll load the model via Assimp and translate it to multiple Mesh objects we've created in the [previous](https://learnopengl.com/Model-Loading/Mesh) chapter.

## Without further ado, I present you the class structure of the Model class:

## **public class Model**

## **{**

## **// model data**

## **private List<Mesh> meshes;**

## 

## **// Constructor, expects a filepath to a 3D model.**

## **public Model(string path)**

## **{**

## **LoadModel(path);**

## **}**

## **public void Draw();**

## **private void LoadModel(string path);**

## **private void ProcessNode(Node node, Scene scene, Matrix4x4 parentTransform)**

## **private Mesh ProcessMesh(AssimpMesh mesh, Matrix4x4 transform);**

## **}**

## The Model class contains a List of Mesh objects and requires us to give it a file location in its constructor then loads the file right away. The private functions are all designed to process a part of Assimp's import routine and we'll cover them shortly.

## The Draw function is nothing special and basically loops over each of the meshes to call their respective Draw function:

## **public void Draw()**

## **{**

## **for (int i = 0; i < meshes.Count; i++)**

## **meshes[i].Draw();**

## **}**

## **Importing a 3D model into OpenGL**

## To import a model and translate it to our own structure, we first need to include Assimp namespace and specify that we want to use AssimpMesh instead of Mesh to avoid confusion between our Mesh class and so that the compiler will know which one we intend:

## using Assimp; using AssimpMesh = Assimp.Mesh;

## Within the constructor we use Assimp to load the model into a data structure of Assimp called a Scene object. You may remember that this is the root object of Assimp's data interface. Once we have the scene object, we can access all the data we need from the loaded model.

First we need to create an importer object:

**AssimpContext importer = new AssimpContext();**

Then we have the option to create a logging callback function that gets called during the ImportFile() method. When we run the Attach() method on the LogStream, messages are passed to the function that can be written to the debug console or logged another way. These messages give us information about the import steps and tell us if errors have occurred.

## 

## **LogStream logstream = new LogStream(delegate (String msg, String userData) { Debug.WriteLine(msg);**

## **}); logstream.Attach();**

With the importer and logstream set up, we can import the scene using the ImportFile() function:

## 

## **Scene scene = importer.ImportFile(path, PostProcessSteps.Triangulate);**

## The function expects a file path and several post-processing options as its second argument. Assimp allows us to specify several options that forces Assimp to do extra calculations/operations on the imported data. By setting PostProcessSteps.Triangulate we tell Assimp that if the model does not (entirely) consist of triangles, it should transform all the model's primitive shapes to triangles first. The PostProcessSteps.FlipUVs flips the texture coordinates on the y-axis where necessary during processing (you may remember from the [Textures](https://learnopengl.com/Getting-started/Textures) chapter that most images in OpenGL were reversed around the y-axis; our texture class already flips our textures, so we will be leaving out this post processing option). A few other useful options are:

## PostProcessSteps.GenerateNormals: creates normal vectors for each vertex if the model doesn't contain normal vectors.

## PostProcessSteps.SplitLargeMeshes: splits large meshes into smaller sub-meshes which is useful if your rendering has a maximum number of vertices allowed and can only process smaller meshes.

## PostProcessSteps.OptimizeMeshes: does the reverse by trying to join several meshes into one larger mesh, reducing drawing calls for optimization.

## Assimp provides a great set of postprocessing options and you can find all of them [here](http://assimp.sourceforge.net/lib_html/postprocess_8h.html) (this is the C++ libraries, but you can pretty easily find a match within the PostProcessSteps enum). Loading a model via Assimp is (as you can see) surprisingly easy. The hard work is in using the returned scene object to translate the loaded data to an array of Mesh objects.

## The complete constructor is listed here:

## **public Model(string path)**

## **{**

## **// Create a new importer**

## **AssimpContext importer = new AssimpContext();**

## 

## **// We can define a logging callback function that receives messages during the ImportFile method and print them to the console.**

## **// These give information about which step is happening in the import such as:**

## **// "Info, T18696: Found a matching importer for this file format: Autodesk FBX Importer."**

## **// or it can give you important error information such as:**

## **// "Error, T18696: FBX: no material assigned to mesh, setting default material"**

## **// Note that in order to see the messages, you must temporarily change your project to be a console application or log the messages another way**

## **LogStream logstream = new LogStream((String msg, String userData) =>**

## **{**

## **Console.WriteLine(msg);**

## **});**

## **logstream.Attach();**

## 

## **// Import the model into managed memory with any PostProcessPreset or PostProcessSteps we desire.**

## **// Because we only want to render triangles in OpenGL, we are using the PostProcessSteps.Triangulate enum**

## **// to tell Assimp to automatically convert quads or ngons into triangles.**

## **Scene scene = importer.ImportFile(path, PostProcessSteps.Triangulate);**

## 

## **// Check for errors**

## **if (scene == null || scene.SceneFlags.HasFlag(SceneFlags.Incomplete) || scene.RootNode == null)**

## **{**

## **Console.WriteLine("Unable to load model from: " + path);**

## **return;**

## **}**

## 

## **// Create an empty list to be filled with meshes in the ProcessNode method**

## **meshes = new List<Mesh>();**

## 

## **// Set the scale of the model. Ideally, the creator of the model would set the initial scale**

## **// and then any further scaling of each instance would take place in the "model" matrix that is passed to the shader.**

## **float scale = 1 / 200.0f;**

## **Matrix4 scalingMatrix = Matrix4.CreateScale(scale);**

## 

## **// Process ASSIMP's root node recursively. We pass in the scaling matrix as the first transform**

## **ProcessNode(scene.RootNode, scene, scalingMatrix);**

## 

## **// Once we are done with the importer, we release the resources since all the data we need**

## **// is now contained within our list of processed meshes**

## **importer.Dispose();**

## **}**

## 

## After we load the model, we check if the scene and the root node of the scene are not null and check one of its flags to see if the returned data is incomplete. If any of these error conditions are met, we write the error to the console and return.

## If nothing went wrong, we set the meshes to be a new empty list. For now, we define a scale (in the future, we would want to give a scale to individual instances of a model with the model matrix that is passed to the shader instead during the initial loading). We then define a scaling Matrix4 using Matrix4.CreateScale(scale) that simply has the scale value along the diagonal, except for a 1 in the bottom right corner.

## Next, we want to process all of the scene's nodes. We pass the first node (the root node) to the recursive ProcessNode function. Because each node (possibly) contains a set of children we want to first process the node in question, and then continue processing all the node's children and so on. This fits a recursive structure, so we'll be defining a recursive function. A recursive function is a function that does some processing and recursively calls the same function with different parameters until a certain condition is met. In our case the exit condition is met when all nodes have been processed.

## As you may remember from Assimp's structure, each node contains a set of mesh indices where each index points to a specific mesh located in the scene object. We thus want to retrieve these mesh indices, retrieve each mesh, process each mesh, and then do this all again for each of the node's children nodes. The content of the ProcessNode function is shown below:

**private void ProcessNode(Node node, Scene scene, Matrix4 parentTransform)**

**{**

**// Multiply the transform of each node by the node of the parent, this will place the meshes in the correct relative location**

**Matrix4 transform = node.Transform.ConvertAssimpMatrix4() \* parentTransform;**

**// Process each mesh located at the current node**

**for (int i = 0; i < node.MeshCount; i++)**

**{**

**// Nodes don't actually carry any of the mesh data, but rather give an index to the corresponding Mesh**

**// within the scene.Meshes List. The Nodes form the hierarchy of the model so that we can establish**

**// parent-child relationships, which are important for passing along transformations.**

**AssimpMesh mesh = scene.Meshes[node.MeshIndices[i]];**

**meshes.Add(ProcessMesh(mesh, transform));**

**}**

**for (int i = 0; i < node.ChildCount; i++)**

**{**

**ProcessNode(node.Children[i], scene, transform);**

**}**

**}**

## We first multiply the local transform of the mesh by the cumulative transform from its parent (Note that initially this is just the scaling matrix and would need to be the identity matrix if we were not scaling here). We then check each of the node's mesh indices and retrieve the corresponding mesh by indexing the scene's Meshes array. The returned mesh is then passed to the ProcessMesh function that returns a Mesh object that we can store in the meshes list.

## Once all the meshes have been processed, we iterate through all of the node's children and call the same ProcessNode function for each of its children. Once a node no longer has any children, the recursion stops.

## A careful reader may have noticed that we could forget about processing any of the nodes and simply loop through all of the scene's meshes directly, without doing all this complicated stuff with indices. The reason we're doing this is that the idea for using nodes like this is that it defines a parent-child relation between meshes. By recursively iterating through these relations, we can define certain meshes to be parents of other meshes.

## An example use case for such a system is when you want to translate a car mesh and make sure that all its children (like an engine mesh, a steering wheel mesh, and its tire meshes) translate as well; such a system is easily created using parent-child relations.

## This is important because our test backpack model has parent-child relationships that are needed to define the transformation of each child mesh. This is why it is generally recommended to stick with this approach for whenever you want extra control over your mesh data. After all these node-like relations are defined by the artists who created the models.

## The next step is to process Assimp's data into the Mesh class from the previous chapter.

### **Assimp to Mesh**

## Translating an AssimpMesh object to a Mesh object of our own is not too difficult. All we need to do is access each of the mesh's relevant properties and store them in our own object. The general structure of the processMesh function then becomes:

**private Mesh ProcessMesh(AssimpMesh mesh, Matrix4x4 transform)**

**{**

**List<Vertex> vertices = new List<Vertex>();**

**List<int> indices = new List<int>();**

**Matrix4 inverseTransform = Matrix4.Invert(transform);**

**for (int i = 0; i < mesh.VertexCount; i++)**

**{**

**Vertex vertex = new Vertex();**

**// process vertex positions, normals and texture coordinates**

**[...]**

**vertices.Add(vertex);**

**}**

**// process indices**

**[...]**

**return new Mesh(vertices.ToArray(), indices.ToArray());**

**}**

## Processing a mesh begins with retrieving all the vertex data and the mesh's indices (at this point we could also load in any texture data, but for simplicity, we will add our texture a different way). The processed data is stored in one of the two lists and from those a Mesh is created and returned to the function's caller.

## Retrieving the vertex data is pretty simple: we define a Vertex struct that we add to the vertices array after each loop iteration. We loop for as many vertices there exist within the mesh (retrieved via mesh.VertexCount). Within the iteration we want to fill this struct with all the relevant data. For vertex positions this is done as follows:

**Vertex vertex = new Vertex();**

**// Positions**

**Vector3 position = mesh.Vertices[i].ConvertAssimpVector3();**

**Vector3 transformedPosition = Vector3.TransformPosition(position, transform);**

**vertex.Position = transformedPosition;**

## First we convert the Assimp position Vector3D into an OpenTK Vector3 using an extension method within an Extensions class:

**public static class Extensions**

**{**

**public static Vector3 ConvertAssimpVector3(this Vector3D AssimpVector)**

**{**

**// Reinterpret the assimp vector into an OpenTK vector.**

**return Unsafe.As<Vector3D, Vector3>(ref AssimpVector);**

**}**

**}**

## Then we apply the mesh transformation to the position using OpenTK’s Vector3.TransformPosition(). Then we simply set the Position member of the new Vertex to be this transformed position.

## Assimp calls their vertex position array Vertices which isn't the most intuitive name.

## The procedure for normals should come as no surprise now:

**// Normals**

**if (mesh.HasNormals)**

**{**

**Vector3 normal = mesh.Normals[i].ConvertAssimpVector3();**

**Vector3 transformedNormal = Vector3.TransformNormalInverse(normal, inverseTransform);**

**vertex.Normal = transformedNormal;**

**}**

One thing of note: as mentioned in the Basic Lighting chapter, normals should not be multiplied by the transformation matrix like the vertex position. We instead use the “the transpose of the inverse of the upper-left 3x3 part of the model matrix'. We can do this fairly easily using two OpenTK functions: Matrix4.Invert() and Vector3.TransformNormalInverse(). Inverting a matrix is an expensive operation, so we only want to do this once for each mesh as shown in the ProcessMesh() method:

**private Mesh ProcessMesh(AssimpMesh mesh, Matrix4x4 transform)**

**{**

**[...]**

**Matrix4 inverseTransform = Matrix4.Invert(transform);**

**[...]**

**}**

## Texture coordinates are roughly the same, but Assimp allows a model to have up to 8 different texture coordinates per vertex. We're not going to use 8, we only care about the first set of texture coordinates. We'll also want to check if the mesh actually contains texture coordinates (which may not be always the case):

**if (mesh.HasTextureCoords(0)) // does the mesh contain texture coordinates?**

**{**

**Vector2 vec;**

**vec.X = mesh.TextureCoordinateChannels[0][i].X;**

**vec.Y = mesh.TextureCoordinateChannels[0][i].Y;**

**vertex.TexCoords = vec;**

**}**

**else vertex.TexCoords = new Vector2(0.0f, 0.0f);**

## Note that we did not want to transform the UV coordinates since they don’t reside in 3D space like the positions and normals. The vertex struct is now completely filled with the required vertex attributes and we can add it to the back of the vertices list at the end of the iteration. This process is repeated for each of the mesh's vertices.

**vertices.Add(vertex);**

### **Indices**

## Assimp's interface defines each mesh as having an array of faces, where each face represents a single primitive, which in our case (due to the PostProcessSteps.Triangulate option) are always triangles. A face contains the indices of the vertices we need to draw in the correct order for its primitive. So if we iterate over all the faces and store all the face's indices in the indices vector we're all set:

**for (int i = 0; i < mesh.FaceCount; i++)**

**{**

**Face face = mesh.Faces[i];**

**for (int j = 0; j < face.IndexCount; j++)**

**indices.Add(face.Indices[j]);**

**}**

## After the outer loop has finished, we now have a complete set of vertices and index data for drawing the mesh via GL.DrawElements. However, we still need to deal with adding some materials.

### **Material**

Assimp scenes can contain material information, but for simplicity we will just be applying a texture similar to how we did it in the Textures chapter.

## Some versions of Assimp tend to load models quite slow when using the debug version and/or the debug mode of your IDE, so be sure to test it out with release versions as well if you run into slow loading times.

## You can find the complete source code of the Model class [here](https://learnopengl.com/code_viewer_gh.php?code=includes/learnopengl/model.h).

# **No more containers!**

## So let's give our implementation a spin by actually importing a model created by genuine artists, not something done by the creative genius that I am. Because I don't want to give myself too much credit, I'll occasionally allow some other artists to join the ranks and this time we're going to load this amazing [Survival Guitar Backpack](https://sketchfab.com/3d-models/survival-guitar-backpack-low-poly-799f8c4511f84fab8c3f12887f7e6b36) by Berk Gedik. Note that there's a few extra texture types we won't be using yet.

Within the OnLoad() function we will need to load the model, texture, and shader. We declare a Model object and pass in the model's file location.

**\_backPack = new Model("Resources/Backpack/Survival\_BackPack\_2.fbx");**

**\_backPackTexture = Texture.LoadFromFile("Resources/Backpack/1001\_albedo.jpg");**

**\_backPackShader = new Shader("Shaders/shader.vert", "Shaders/shader.frag");**

Within the OnRenderFrame function we set up the uniforms for the shader, including binding the texture. We then call the Draw function for the model:

**\_backPackShader.Use();**

**\_backPackShader.SetMatrix4("model", Matrix4.Identity);**

**\_backPackShader.SetMatrix4("view", \_camera.GetViewMatrix());**

**\_backPackShader.SetMatrix4("projection", \_camera.GetProjectionMatrix());**

**\_backPackTexture.Use(TextureUnit.Texture0);**

**\_backPackShader.SetInt("texture0", 0);**

**\_backPack.Draw();**

## That’s it! We are now using simple shaders where the fragment shader only outputs the object's diffuse texture, the result looks a bit like this:

## 

## You can find the complete source code [here](https://learnopengl.com/code_viewer_gh.php?code=src/3.model_loading/1.model_loading/model_loading.cpp).

## We can also get more creative and introduce point lights to the render equation as we learned from the [Lighting](https://learnopengl.com/Lighting/Light-casters) chapters and together with specular maps get amazing results (covered in next section):

## 

## Even I have to admit that this is maybe a bit more fancy than the containers we've used so far. Using Assimp you can load tons of models found over the internet. There are quite a few resource websites that offer free 3D models for you to download in several file formats. Do note that some models still won't load properly, have texture paths that won't work, or are simply exported in a format even Assimp can't read.

## **Further reading**

## [How-To Texture Wavefront (.obj) Models for OpenGL](https://www.youtube.com/watch?v=4DQquG_o-Ac): great video guide by Matthew Early on how to set up 3D models in Blender so they directly work with the current model loader (as the texture setup we've chosen doesn't always work out of the box).

## 