Rendering an Interactive Wooden Artist's Mannequin

Exploring Inverse Kinematics and Procedural Wood Shading

Sean Carroll, Amber Hillhouse, Brian Kutsop, Youenn Paris, Daniel Sperling, Charles Yan

December 15, 2014

1 Introduction

The purpose of our project was to create a program that allows the user to ultimately create a high-quality rendered image of a wooden artist's mannequin in a position of their choice.

This program consists of two main parts: an interface where the user can interact with and position the model dynamically through the use of triangulation and cyclic coordinate descent (CCD), and a wood shader that generates the image. Once the mannequin has been positioned into the correct pose, an accurate wood rendering can be generated and saved.

2 Features and Capabilities

Cyclic Coordinate Descent (Key C): This rotates all of the current joint's parent joints through to the root (body center) of the mannequin. The motion is less centralized but more realistic looking.

Triangulation (Key V): This allows the user to have more local control by rotating only the previous 2 joints in an attempt to solve for the exact triangle between the three joints. Because the middle joint (think "elbow" for the arm) has a circular-shaped degree of freedom around the axis between the top and lower joint (think shoulder and wrist), the behavior tends to be more unpredictable than CCD.

Object/World Coordinates (Key N for Object, Key B for World): This makes the CCD or triangulation try to follow the object or world coordinates of the end effector as closely as possible when dragged.

Number of CCD joints effected (Key 0 for all joints through root, Any number key for # of joints): When in CCD mode, this allows the user to specify how many earlier joints are effected by the CCD.

Wood Rendering (Key F4): This uses ray tracing to place a realistic wood shading on the mannequin, and saves the generated image as a .png file in the project's data/scenes/ directory.

3 Inverse Kinematics

Mannequin Mesh Hierarchy: The full mannequin mesh was created by: (1) Downloading the full mesh from a free open source Blender project,[1] (2) Saving each of the individual body parts with the mesh vertices in object space, and (3) Creating an XML that contains the correct obect hierarchy and initial translations/rotations of the joints. The meshes are saved in the /data/meshes/mannequin directory, and the XML is saved in the /data/scenes directory.

Joint Objects: In order to represent joints, we created subclasses of the SceneObject and RenderObject classes, classes SceneJoint and RenderJoint respectively. The common/Scene.java file was edited in order to create SceneJoints from an XML ⟨joint⟩ tag, and gl/RenderTreeBuilder.java was edited to create RenderJoints from SceneJoints.

Manipulator Controller: Because the orientation of the end effector changes as we rotate the joints to perform the translations, and we wanted the translations to move along the original axis when the joint was chosen instead of the newly rotated axis, we added a boolean field called "recalc_axes" to gl/manip/manipulator.java that is set to be true when a joint is selected, and false once the axis of translation is calculated. This prevents the end effector from moving in circular motions.

Triangulation: The triangulation of the joints is calculated by: (1) Calculating the new desired position of the end effector by translating along the selected manipulator, (2) Getting the positions of the joints in the object space of the top joint, (3) Calculating the triangle side lengths and angles for the 3 local joints, (4) Taking the cross product of the old and new positions of the end effector to determine the axis of rotations around each of the joints, and (5) Using quaternions to calculate the rotation matrices around the top and middle joints.

CCD: CCD is implemented according to the following steps: For each iteration of CCD, and for each joint being considered from end effector through root: (1) Calculate the current position of the end effector using any temporarily calculated rotations, (2) Determine the difference between the current end position and desired end position, and stop iterating if we are "close enough", (3) Get the positions of

[1]http://jaxmp.deviantart.com/art/Blender-Art-Mannequin-89266968

the joints in the object space of the current joint being rotated, (4) Use quaternions to calculate the new rotations, (5) Store all currently calculated rotations in the RenderJoint ”temp_transformation” field so that new positions are not rendered during the calculation process, and (6) Set all joint transformations equal to their temporarily saved transformations to set the final positions. The current number of iterations is set at 5, and the implementations are located in ManipControl.java.

4 Wood Rendering

4.0.1 Wood Coloring

In order to create realistic wood, we developed a model for calculating growth ring positions in 3D as if the model was being cut out of a large piece of wood, instead of applying a 2D texture. Calculating the rings in 3D is more realistic than simply using a 2D texture because it prevents the texture from being stretched or skewed in ways that would not happen if one was to carve something out of a piece of wood. Furthermore, the 3D model allows wood to be rendered on any surface - with only minor parameter changes to account for the size of what is being modeled - without the need for UV coordinates, which change from object to object.

To create rings, we sampled a light and dark brown from a real picture of wood, and developed a function that, for each “ring”, would go gradually from light to dark, more quickly at the end, and then extremely quickly back to light, or the beginning of a new ring. We developed and found success with the following formula:

$$\text{color}(f,t) = \frac{f + \cos(2 * \pi * (1-t)^{0.25})}{(f+1)} \tag{1}$$

where f is a scaling factor and $0 \leq t \leq 1$, which creates the following graph:

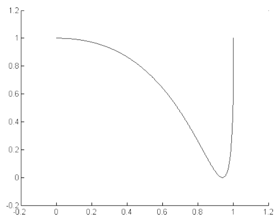

Figure 1: Graph of (1)

An important factor in generating more realistic wood is randomness. We used Perlin Noise and scaled this noise by a randomly generated scaling factor in order to create randomness in the shape of the rings and to create random particle imperfections on the edges of rings.

4.0.2 Subsurface Reflections

In order to generate a realistic rendering, our group decided to integrate with the ray tracing portion of the project. This gave the ability for shadowing, anti-aliasing, and much more complicated reflection.

We implemented subsurface reflection based on the paper ”Measuring and Modeling the Appearance of Finished Wood” by Marschner, Westin, Arbree, and Moon. This paper details the complicated nature of glossy wooden surfaces, where a portion of incoming light is reflected off of the surface while the remainder is refracted into the material, diffusely reflects off of a portion, and also reflects directionally based on the direction of the wooden fibers in that area. This produces the ”sheen” effect seen on nice, polished wood and is a clear differentiation between real wood and the wood that is frequently used for renderings.

Mathematically, our group followed the formula:

$$f_r(v_i, v_r) = f_s(v_i, v_r) + T_i T_r(\rho_d + f_f(u, v_i, v_r) \tag{2}$$

where

$$f_f(u, v_i, v_r) = k_f \frac{g(B, \psi_h)}{cos^2(\psi_d/2)} \tag{3}$$

and

$$\psi_i = sin^{-1}(s(v_i) \cdot u) \tag{4}$$
$$\psi_r = sin^{-1}(s(v_r) \cdot u) \tag{5}$$
$$\psi_d = \psi_r - \psi_i \tag{6}$$
$$\psi_h = \psi_r + \psi_i \tag{7}$$

As explained in the paper, the first equation separates the reflection into surface (f_s) and subsurface ($f_f + \rho_d$) reflection, where ρ_d is diffuse subsurface reflection and f_f is the fiber reflection, attenuated by $T_i T_r$ representing entering and exiting the subsurface material.

Fiber reflection is represented by a the specular coeficient k_f, and a Gaussian term and cosine term around the reflection of the viewing angle with the fiber angle, generally represented by ψ. $s(v)$ represents the refraction of the viewing angle into the surface.

Once we implemented subsurface refection, we varied the fiber direction on a per-object basis in the xml, and used the fiber direction to also determine the general direction of the pattern of the rings in the wood to mimic realistic wooden behavior.

5 Integration

The framework used to interact with the mesh was not directly compatible with the framework for the ray tracer. There existed a function to export an XML file with the positions of the mesh on the GUI, but it was not the correct format for the ray tracer to parse, as the Scene class in the rasterization code is vastly different than the Scene class in the Raytracer. We parsed SceneObjects from one scene to Surfaces in the other.

The camera provided its own challenge, as in one form it was represented as a matrix from the center and a pair of z planes, and in the ray tracer it was a view direction, position, up, and projection distance. Using matrix multiplication based on the original world position we were able to move the camera to its correct location. Once this was completed, rendered images could be produced from the viewed scene.