14.6 Construction and Animation via Hierarchical Modeling

a) Motivation

Note: Throughout section 14.6, we will refer you to activities in Lab #4 ("Hierarchical Modeling"). We strongly recommended that you perform the lab activities in tandem with your reading of this section.

What’s a desert without camels? In this section, we will design a simplistic robotic camel with moving parts that allow us to produce simple animations. The sequence of screenshots below show some frames from a fixed-camera animation sequence that is the target of the model-construction activities described in this section and available for interactive experimentation in the online lab. This section builds on the modeling and animation techniques you first encountered with the clock example in Chapter 2. While the XAML code examples below are, of course, particular to WPF, the techniques illustrated here for composing and animating hierarchical models have wide applicability in computer graphics regardless of the modeling and rendering APIs of a particular graphics platform.

The animation begins (frame #1) with the camel in its “canonical” position, standing on the desert directly centered on the origin. He levitates skyward for a few seconds (see frame #2) via rockets on his feet (use your imagination!); as he rises, his knees begin to bend slowly towards a 45-degree angle (see frame #2). As soon as his rockets really kick in, he starts moving at great speed (see frame #3).

When faced with the need to specify an object of non-trivial complexity like our robotic camel, a developer must consider modularizing the specification of the geometry by dividing the model into subcomponents.

Although there is theoretically no end to the complexity that can be represented by a single mesh specification – and thus it is possible to specify this camel with a single GeometryModel3D element – there are several shortcomings to be found in such a monolithic design strategy:

- In WPF, materials are specified at the level of GeometryModel3D, not at the level of individual triangles in the mesh. So if you want the materials to vary (e.g. to have the camels’ feet be a special color), you must divide your model into subcomponents.
• If there are certain subparts that are found multiple times in the larger model (e.g., the four near-identical legs of a camel), it is convenient to be able to define such parts once and then "clone" them as needed to create the larger model. Reusability of components, customized via parameters, is as fundamental to computer graphics modeling and rendering as it is to software construction.

• The use of sub-components facilitates the animation/motion of sub-parts. If a complex object is defined via a single mesh, independent movement of subparts requires editing the mesh. If the design is modular, however, movement can be implemented via transforming the independent subparts (e.g., to achieve the camel’s “bended knee” appearance, we are simply applying a rotation to its “lower leg” subcomponent about a pivot point (axis of rotation) between the upper and lower leg).

• WPF’s support for mouse/pen interaction with a 3D scene requires the division of complex objects into subcomponents, to simplify the customization of interaction responses. For example, if you want interaction with a camel’s leg to respond in a manner quite distinct from interaction with its torso, you should define those parts in distinct subcomponents.

• Editing a single-mesh complex model is extremely difficult due to interdependencies between the various parts – e.g. a simple act like extending the height of the camel legs would, as a side-effect, change the coordinates of all the mesh coordinates in the camel’s head and trunk. But when a model is defined using a hierarchy of independent sub-parts, the geometry of a sub-component can be edited in isolation and often there is no need to touch the geometry of other parts (the exception being a geometric change to a part that is intended to connect to another sub-component, e.g. a change to the socket shape of a ball-and-socket joint).

b) Top-Down: Modular Component Design

One strategy for designing a complex model for animation applications is to approach the model “as a whole”, focusing on determining the locations of “joints” at which movement might be desired. For example, the animation requirements for our robotic camel might require only knee joints, hip joints, and a simulated “neck” for head rotation. One example of each of these three joint1 types is highlighted in this figure...

After identifying the simple joints used in this animation, we can diagram the hierarchy of component types; in our example below, grey rectangles represent the composite groups (Model3DGroup) and blue ones represent the atomic mesh objects (GeometryModel3D).

---

1 Here, our use of the term “joint” is informal and simply identifies locations at which we might want to implement an axis of rotation on a subcomponent to simulate a biological joint or a construction hinge. In sophisticated animation technologies, the term “joint” carries much more meaning and is often an actual object (distinct from the model’s subcomponents) with appearance and behaviors derived from knowledge in the fields of physics and biology. See Chapter ____ for a brief discussion of this specialized topic.
The joints are represented in this diagram as red diamonds. Each diamond lies on a line connecting parent $P$ to subcomponent $S$, and represents a place where the instantiation of $S$ would include a transformation “hook” to implement the movement type that the joint represents.

The above diagram is NOT an “inheritance” (class) hierarchy. Rather, it is a “composition-type” hierarchy that lists the types of objects that are needed to build an object of a higher-level type. E.g., in order to construct a Leg, we must accumulate subcomponents of type Thigh and LowerLeg.

Note that because of its focus on “types” instead of “instances of types”, it does not give information about the number of instances (copies of a type) needed for construction. An “instance hierarchy” is thus of value during implementation:

Each block in an instance hierarchy represents a particular copy of a component type. Thus, instead of a box representing the “Leg” type, (which is a component type), we have four uniquely named copies of the component hierarchy that is the canonical leg “template”.\(^2\)

\(^2\) Again, the similarity to the clock application in the WPF-2D chapter should be noted. The canonical clock-hand template which was instantiated twice (with varying “instance transforms” for sizing/positioning) to form the hour/minute hands is equivalent to the “leg” component type which will be instantiated four times (also with varying transforms for positioning) to form the camel’s legs.
It should be noted that the determination of the component/mesh hierarchy should not be driven solely by animation needs; for example, the need for varying materials also drives the modularization process, since only one material can be assigned to any given mesh object.

c) Bottom-up: Component Construction and Model Composition

Now we demonstrate how XAML can be used to implement the model’s construction. The order is bottom-up: first generating the atomic components (foot, shin, etc.) and then doing the compositing to create the higher-level components (starting with intermediate ones like lower-leg, and moving up the hierarchy).

IMPORTANT: The technique shown here is optimal only for prototyping and proof-of-concept. It uses XAML for rapid edit/execute prototyping cycles, and it supports define-once/use-many specification for the meshes of the atomic components; however, it does not provide for define-once/use-many techniques for composite components. Thus, this is not the technique one would use for a scene involving dozens or hundreds of copies of a model. A hybrid XAML+code approach — in which the components are instantiated by procedural code (e.g. in C#) via references to resources specified in XAML — is optimal for scenes involving a large amount of reuse at all levels of the model hierarchy (composites and meshes).

The activities involved in bottom-up construction are summarized in this table:

<table>
<thead>
<tr>
<th>Intended goal</th>
<th>Where</th>
<th>WPF Element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify the geometry of an atomic component type (e.g. “Foot”)</td>
<td>Resource section</td>
<td>MeshGeometry3D</td>
</tr>
<tr>
<td>Instantiate an atomic component</td>
<td>Inside the content of a viewport, as a direct child of the Model3DGroup representing its direct composite parent.</td>
<td>GeometryModel3D element. Name attribute provides a unique ID useful for animation and pick correlation. Geometry attribute points to the corresponding MeshGeometry3D resource. GeometryModel3D.Transform child element can be used to specify an “instance transform” and/or joint transform. Model3DGroup element. Name attribute provides a unique ID useful for animation and pick correlation. Model3DGroup.Transform child element, as described above.</td>
</tr>
<tr>
<td>Instantiate a composite node</td>
<td>Inside the content of a viewport, as a direct child of the Model3DGroup representing its direct composite parent.</td>
<td>Model3DGroup element. Name attribute provides a unique ID useful for animation and pick correlation. Model3DGroup.Transform child element, as described above.</td>
</tr>
</tbody>
</table>
### 14.5(c).1 Specifying Atomic Component Geometries

The geometric design of each atomic component can be considered to be an independent task/project in the development effort. In some respects, the design of a particular atomic component should be independent of other components, e.g.:

- Should be specified in its own coordinate system.
- For convenience, can/should be at/near the origin of its own coordinate system (e.g., resting on one of the three primary planes, centered on one of the primary axes, etc.)
- For convenience, can/should be oriented in a canonical way (e.g. parallel to one of the axes).

In other respects, to ensure connectivity, the design of a component needs to take into account the “big picture”, e.g.:

- The compositing effort is greatly simplified if the sizes of the atomic components are consistent. By this, we mean that translation and rotation transforms (to place the parts in proper adjacency relative to each other) should be sufficient for composition, in that the sizes will be already “correct” once the parts have been moved into position. In our example here, we use consistent sizes; the foot is about 19 units high and the shin is about 30 units high, and no stretching/compressing (scaling) actions are necessary in compositing the lower leg from these two atoms; similarly, the thigh is consistently sized so the full leg can be built via simple translation transforms of the thigh and the lower leg.
- The compositing effort can also be simplified if the orientations of the atomic parts are consistent. In our example here, we have designed both the foot and the shin so they already have correct orientation for the situation in which the camel is standing on the XZ plane. This is a suggestion, not a requirement, since rotations can be used to correct any inconsistencies in orientation, and parts can be resized if needed.

Note that manual generation of the mesh specifications is not a realistic way to generate non-trivial objects. To generate the atomic components, one would typically use a 3D-modeling application that exports XAML mesh specifications.

The table below shows information about geometries of our first two atomic components:

| MeshGeometry3D element defined in the resource section, named “RSRCmeshFoot”. |
| 744 vertices. |
| 420 triangles. |
| Dimensions: |
| 30 units in diameter on both X and Z axes. |
| 18.75 units in height on Y axis. |
14.5(c).2  Instantiating an atomic component

We have now created a library of object “templates,” but we haven’t created anything truly visible yet.

To see one of our atomic components rendered in a 3D scene, e.g., to test the accuracy of its mesh specification, we can instantiate it directly inside the viewport’s content. Instantiation of such a “standalone” component is done via creation of a GeometryModel3D element to be placed in the Model3DGroup element that specifies the viewport’s content. This technique is identical to that which was used in our pyramid example.

The XAML shown below adds an instantiation of our foot atomic component to the viewport which we had setup in section 1.2(b) earlier in this chapter:

```xml
<ModelVisual3D.Content>
  <Model3DGroup>
    <!-- Specify lights here. -->
    <GeometryModel3D Geometry='{StaticResource RSRCmeshFoot}' Material=... />
  </Model3DGroup>
</ModelVisual3D.Content>
```

Note that the instantiated foot is not being transformed, so it will appear at the origin of the world coordinate system of the scene being specified for this viewport.

Enter the online lab, and use the “Model:” listbox, along with the turntable feature, to instantiate and examine the atomic components of the camel.
Instantiating a composite component

Just as each atomic component has its own coordinate system, each composite component also has its own coordinate system. But whereas an atomic component has inherent geometry, a composite’s content is specified via the instantiation of one or more subcomponents, which are “copied” by WPF into the composite’s own coordinate system.

The subcomponents can be atomic, or can be composites themselves. In our first example of a composite, we look at the lower leg, which is a composite made up of two atoms.

COMPOSITING TO CREATE THE LOWER LEG

We instantiate a composite node, such as the LowerLeg component, by constructing an Model3DGroup element (representing the entire component) and populating it with instantiations of its subcomponents. Here is a “first draft” of our camel’s lower leg:

```xml
<Model3DGroup x:Name="LowerLeg">
  <GeometryModel3D Geometry='{StaticResource RSRCmeshFoot}' Material=... />
  <GeometryModel3D Geometry='{StaticResource RSRCmeshShin}' Material=... />
</Model3DGroup>
```

Here again, for testing purposes, one can instantiate this composite component directly into the viewport’s content in order to test it in a “standalone” fashion.

In the online lab, select the model “Lower leg (shin+foot)”.

Note that the two atomic components have been unintentionally merged. Why did this occur? Each atomic component was defined conveniently at the origin of its own coordinate system. But now that they have both been instantiated onto a single coordinate system, they are occupying each other’s space... and our lower leg is an unwanted union of the two objects.
Introduction to WPF 3D

Obviously, our goal is that the shin be connected to the ankle part of the foot, as shown here:

The solution is to use instance transforms\(^3\) to help position subcomponents brought into a composite. In this composition of the lower leg, an instance transform is needed for the shin, but not for the foot. The shin’s placement transform will be a simple translate in the positive \(Y\) direction.

**In the online lab, use the hierarchy viewer/editor to add a transform to the shin to repair the lower-leg composite.**

Here is our second draft of the XAML specification for this composite component (with previously presented XML fragments rendered in grey text):

```xml
<Model3DGroup x:Name="LowerLeg">
  <GeometryModel3D Geometry='{StaticResource RSRCmeshFoot}' Material=… />
  <GeometryModel3D Geometry='{StaticResource RSRCmeshShin}' Material=… >
    <GeometryModel3D.Transform>
      <TranslateTransform3D OffsetY='14'/><!-- Raise the shin. -->
    </GeometryModel3D.Transform>
  </GeometryModel3D>
</Model3DGroup>
```

**COMPOSITING TO CREATE THE FULL LEG**

Let’s continue our bottom-up implementation by going up to the next level. We need to add the thigh component to the lower leg, but there is a new “twist” to this composite because it includes a joint: the lower leg is intended to be rotatable to simulate a bendable knee where the thigh meets the lower leg.

We have already learned that one aspect of creating composites is to attach instance transforms to place subcomponents being brought in. Now we learn of a second type of compositing activity: attaching “joint transforms” to subcomponents in order to provide for mobility.

---

\(^3\) The concept of instance transformation, which is just as relevant in 2D scene constructions, was introduced in section 2.2.5.
Our goal is for the lower leg to look like this:

We create this composite by instantiating both the lower leg and the thigh.

Return to the online lab, and examine the Thigh atomic component by selecting from the list of models.

Then select the model “Whole leg”. The undesired merging of the two subcomponents will be obvious. Repair by adding an instance transform to the thigh to translate it on the $Y$ axis. (If you wish, you can jump straight to our solution by choosing “Whole leg auto-composed” from the list of models.)

Here is the XAML specification of the whole leg with the thigh-raising instance transform in place:

```xml
<!-- Instantiate the whole-leg composite. -->
<Model3DGroup x:Name="Leg">

<!-- Instantiate the lower-leg composite (same XAML shown earlier). -->
<Model3DGroup x:Name="LowerLeg" ... </Model3DGroup>

<!-- Instantiate the thigh. -->
<GeometryModel3D Geometry='{StaticResource RSRCmeshThigh}' Material=...>
    <GeometryModel3D.Transform>
        <TranslateTransform3D OffsetY='37'/>
    </GeometryModel3D.Transform>
</GeometryModel3D>
</Model3DGroup>
```

ADDING A SIMULATED JOINT

The leg we have created is locked into a straight position. By adding a rotation transformation to the lower leg, we can provide a “hook” that animation logic can use to simulate bending at the knee.

The figure below shows the leg in its canonical location at the origin, but with a 37-degree rotation on the knee. (The invisible axis of rotation has been added to this rendered image for clarity.)
Introduction to WPF 3D

The WPF element for expressing 3D rotation requires specification of the axis and the rotation amount. The axis is configured via two parameters: a directional vector (e.g. [1,0,0] representing any line that is parallel to the X axis) and a centerpoint (e.g. the center of the “knee” part of the lower leg, which is (0,36.5,0) in the lower-leg’s coordinate system).

Return to the whole-leg model in the online lab. Implement the simulated knee joint by adding a rotation transform to the lower-leg component. Set the rotation axis as needed, and then use the numeric spinner to change the rotation amount to produce a knee-bend animation.

The XAML for the lower leg – with the joint transformation in place – is as follows:

```xml
<!-- Instantiate the lower-leg composite (same XAML shown earlier) -->
<Model3DGroup x:Name='LowerLeg'>
  <!-- Animation transform for the lower leg's knee "joint" -->
  <Model3DGroup.Transform>
    <RotateTransform3D X='0' Y='36.5' Z='0'>
      <RotateTransform3D.Rotation>
        <AxisAngleRotation3D x:Name='kneeRotationForAnim' Angle='37' Axis='1 0 0'/>
      </RotateTransform3D.Rotation>
    </RotateTransform3D>
  </Model3DGroup.Transform>
  <GeometryModel3D Geometry='{StaticResource RSRCmeshFoot}' Material=... />
  <GeometryModel3D Geometry='{StaticResource RSRCmeshShin}' Material=... >
    <TranslateTransform3D OffsetY='14'/>
    <GeometryModel3D.Transform>
      <!-- Raise the shin -->
    </GeometryModel3D.Transform>
  </GeometryModel3D>
</Model3DGroup>
```

COMPLETING THE CAMEL VIA “CLONING” THE LEG TEMPLATE

The instantiations of the head and trunk are trivial; they are atomic components that simply need to be transformed from their default locations at the origin.
The camel needs four legs. Which leg have we been building during the construction activities we’ve done thus far? One way to look at the earlier part of this section is this: we have been constructing a “template leg” that is not a particular leg but rather is something we can “clone” multiple times to create the actual legs.

In the online lab, choose the “Head + Trunk + Front-left Leg” model to instantiate the head, the trunk, and a complete copy (“clone”) of our leg “template”. Then, add instance transforms as needed to position the head and trunk appropriately. Use the one leg as a guide to help you decide how much to raise the head and trunk above the ground plane. Then, attach the front-left leg to the trunk by sliding the leg along the ground plane as needed.

If you wish to continue the construction, choose “Head + Trunk + 2 front legs” model to add a second cloned leg, and then transform that instance into position as the front-right leg.

In a pure-XAML application, you perform cloning by making a complete copy of the XAML code that specifies the template being cloned. (The copy should not be exact—in particular, you would want to vary the x:Name attributes to ensure the procedural code has the ability to animate each joint in the entire model independently, as described in the next subsection.)

Thus, our complete camel (head, trunk, plus four legs) is specified in pure XAML thusly:

```xml
<Model3DGroup>
  <GeometryModel3D Geometry='{StaticResource RSRCmeshHead}' Material=…>
    <!-- Instance transform for the head would go here -->
  </GeometryModel3D>

  <GeometryModel3D Geometry='{StaticResource RSRCmeshTrunk}' Material=…>
    <!-- Instance transform for the trunk would go here -->
  </GeometryModel3D>

  <!-- Instantiate the front-left leg by "cloning" XAML code -->
  <Model3DGroup x:Name='FrontLeft_Leg'>
    <!-- The instance transform to place the front-left leg would go here -->
    <!-- A complete copy of the XAML for the whole-leg composite model would go here -->
  </Model3DGroup>

  <!-- Instantiate the front-right leg by "cloning" XAML code -->
  <Model3DGroup x:Name='FrontRight_Leg'>
    <!-- The instance transform to place the front-right leg would go here -->
    <!-- A complete copy of the XAML for the whole-leg composite model would go here -->
  </Model3DGroup>

  <!-- Instantiate the rear-left leg, using the same technique -->
  <!-- Instantiate the rear-right leg, using the same technique -->
</Model3DGroup>
```

In the online lab, choose the “Entire camel auto-composed” model to see the finished product, then choose the “XAML” tab to view the complete XAML source code.

---

4 This approach is clearly not scalable and is only for prototyping/instructional purposes. In a typical real-world WPF application, procedural code would be used to instantiate “templates” constructed in XAML.
ANIMATING THE CAMEL MODEL

In section 2.3.1, you were introduced to WPF’s animation features, including the XAML element DoubleAnimation that can be used to force a numeric-type property to travel through a given range of values in a given amount of time. The very same element can be used to act on the properties of 3D objects to achieve dynamics like automatic rotations and translations.

In the online lab, choose the “Entire camel + animation” model to watch the animation sequence that is described in this section.

The “rocket camel” animation sequence implemented in the online lab (and shown via the figure sequence presented earlier) can be specified entirely in XAML, via a “parallel-timeline” storyboard describing a set of three separate animations that for the most part occur simultaneously:

a) Starting immediately upon animation launch, the entire camel model begins to be translated up the Y axis, to cover 50 units in 13 seconds.

b) Starting 3 seconds after the animation launch, the camel’s knees begin to bend at a rate of 45 degrees in 5 seconds. The bending stops when it reaches 45 degrees.

c) Starting 5 seconds after the animation launch, the entire camel begins to move in the positive Z direction, to cover 500 units in 10 seconds.

The structure of the specification, which lies in the XAML application’s resource section, is as shown:

```xml
<Storyboard x:Name='RobotLaunchStoryboard'>
  <ParallelTimeline>
    ... The animation elements that modify property values are here...
  </ParallelTimeline>
</Storyboard>
```

Let’s first implement animation part (a), whose goal is to move the entire camel in the positive-Y direction. Some facts about this animation:

- The target object of the animation is the Model3DGroup that is the root node of the camel model.
- That Model3DGroup object must have a TranslateTransform3D element that acts as the target of the numeric-value manipulation.
- The OffsetY property of the TranslateTransform3D object is the numeric property that will be manipulated.

When first constructing the camel, we did not attach any transformations to the root Model3DGroup, because it was not recognized as a “joint” or as a place where transforms would be needed. But now that we are starting to think about animation, it is clear that it requires a transform so the camel “as a whole” can be moved. We thus supplant the camel’s XAML specification with a named transform that can be referenced by an animation element:

```xml
<Model3DGroup x:Name='Camel'>
  <Model3DGroup.Transform>
    <TranslateTransform3D x:Name='camelTranslationForAnim' />  
  </Model3DGroup.Transform>
</Model3DGroup>
```
Now that we have the element to be manipulated, we can specify the XAML that implements animation part (a):

```xml
<DoubleAnimation
    Storyboard.TargetName="camelTranslationForAnim"
    Storyboard.TargetProperty="OffsetY"
    From="0" To="50"
    Duration="0:0:13"/>
```

**Exercise for the reader:** Show the XAML that would implement animation part (c).

In order to implement part (a), we had to supplant the specification of the camel to create the target for the value manipulation – do you think further modification of the camel specification will be necessary to implement part (c)?

The implementation of part (b) – the bending of the knees – requires modification of the rotation joint that we had already designed into the leg template. Since the leg was cloned four times, we have four targets to manipulate, and thus four distinct DoubleAnimation elements will be necessary. But how are we going to tell a particular DoubleAnimation element to modify a particular knee, e.g. the front-right knee? If the four clones are completely identical, how is it possible to point to a particular knee joint?

In order to provide for dynamics like this animation, the clones must not be 100% identical copies – there must be some way to uniquely identify subcomponents within a particular clone.

The strategy that we use in the online lab is one we call “deep renaming.” When the leg template is being cloned to create a particular leg, each named subcomponent in the template is given a new name via prepending a “clone ID”. For example, the clone ID of “FrontRight_” is used when creating the clone for the front right leg. Thus, in the generated front right leg, the Model3DGroup representing the lower leg is named “FrontRight_LowerLeg”, the AxisAngleRotation3D element that simulates the knee is named “FrontRight_kneeRotationForAnim,” etc.

**EXERCISE FOR THE READER:** You now have all the information you need to create the four DoubleAnimation XAML elements that implement the knee-bending dynamics of part (b) of our animation.

An alternative strategy is one we might call “root renaming.” Consider a cloning operation in which only the root is given a unique name; in this case, the front-right leg would be a Model3DGroup named “FrontRight_Leg” but inside its content there would be no unique name (e.g. all four lower-leg subcomponents would be named “LowerLeg”). In this situation, is it possible to point an animation element to a subcomponent inside a particular clone? Yes it is possible, via WPF’s support for property paths which allow identification of a subcomponent via its relationship to a particular ancestor. Using property paths, it is possible to refer to the front-right knee by specifying a path from the element that is the root of the front-right leg. Property-path syntax is complex and is beyond the scope of this textbook, but you should note that it is a powerful technique that reduces an application’s reliance on defining and maintaining x:Name attributes.

-FIN-