Lab 2: Hierarchical Pose-to-Pose Animation

Published | Edit | ⋮

GPR-450 Advanced Animation Programming
Instructor: Daniel S. Buckstein
Lab 2: Hierarchical Pose-to-Pose Animation

Summary:
In this lab we explore the fundamentals of pose-to-pose animation using skeletal (hierarchical) animation principles: poses, hierarchies, multi-channel animation and forward kinematics. This is a lead-in to project 2 and general character animation.

Submission:
Start your work immediately by ensuring your coursework repository is set up, and public. Create a new main branch for this assignment. ***Please work in pairs (see team sign-up) and submit the following once as a team***:

1. Names of contributors
   e.g. **Dan Buckstein**
2. A link to your public repository online
   e.g. **https://github.com/dbucksteinccorg/graphics2-coursework.git**
   (note: this not a real link)
3. The name of the branch that will hold the completed assignment
   e.g. **lab0-main**
4. A link to your video (see below) that can be viewed in a web browser.
   e.g. <insert link to video on YouTube, Google Drive, etc.>

Finally, please submit a ***5-minute max*** demo video of your project. Use the screen and audio capture software of your choice, e.g. Google Meet, to capture a demo of your project as if it were in-class. This should include at least the following, in enough detail to give a thorough idea of what you have created (hint: this is something you could potentially send to an employer so definitely show off your professionalism and don't minimize it):

- Show the final result of the project and any features implemented, with a voice over explaining what the user is doing.

- Show and explain any relevant contributions implemented in code and explain their purpose in the context of the assignment and course.
- Show and explain any systems source code implemented, i.e. in framework or application, and explain the purpose of the systems; this includes changes to existing source.
- **DO NOT AIM FOR PERFECTION, JUST GET THE POINT ACROSS**. Please mind the assignment rubric to make sure you have demonstrated enough to cover each category.
- **Please submit a link to a video visible in a web browser, e.g. YouTube or Google Drive.**

Objectives:
Furthering the theme of decoupled systems and data, the main goal of this assignment is to create the fundamental data structures for hierarchical pose-to-pose animation. You will continue to think abstractly, this time with the application of skeletal animation to demonstrate the fundamental algorithm of forward kinematics.

Instructions & Requirements:
DO NOT begin programming until you have read through the complete instructions, bonus opportunities and standards, start to finish. Take notes and identify questions during this time. The only exception to this is whatever we do in class.
Using the framework and object- or data-oriented language of your choice (e.g. Unity and C#, Unreal and C++, animal3D and C), complete the following steps:

1. ***Repository setup***: See lab 1 for the complete setup instructions.
    - Check out the 'anim/skeletal' branch from the course repository, it has some stuff you can start with.
2. ***Data structures***: Implement the following decoupled data structures (the importance of decoupling and abstract thinking will be explained throughout) and associated methods:
    - ***Hierarchy node***: The core of all hierarchical data. This is entirely decoupled from any temporal or spatial data; it is strictly used to describe the topology or depth of nodes in a hierarchy (tree data structure). ***Node types that combine this with spatial/temporal information (e.g. Unity's GameObject) are not decoupled***.
        - *Name*: A string of fixed length identifying the node.
        - *Index*: Index in hierarchy (see below).
        - *Parent index*: Index of parent index in hierarchy; should always be less than index (negative if this is a root node, meaning it does not have a parent node).
        - *Constructor/factory/initialize*: set the above data.
    - ***Hierarchy***: Effectively a node pool, however nodes are sorted into a tree by depth. See *a3_Hierarchy* in framework for a complete and functional example.
        - *Nodes*: array of all nodes in the hierarchy, sorted by tree depth (parents are always before children in the array).
        - *Count*: number of nodes in the hierarchy.
        - *Constructor/factory/initialize*: allocate array of nodes, initializing all to default values.

        - *Destructor/release/cleanup*: deallocate array.
    - ***Spatial pose***: A description of a transformation (pose) in space. This will be used to provide the spatial/temporal context for a hierarchy node: it represents the spatial description of a single node at some point in time (i.e. the sample data in a keyframe for a single node).
        - *Transform*: The 4x4 transformation matrix described by the pose, relative to the parent space.
        - *Orientation*: Three elements describing Euler angle orientation relative to the parent space.
        - *Scale*: Three elements describing the scale relative to the parent space.
        - *Translation*: Three elements describing the translation relative to the parent space.
        - *Constructor/factory/initialize*: Assign default or user-provided values for each channel (defaults: transform = identity, rotation channels = 0, scale components = +1, translation channels = 0).
    - ***Hierarchical pose***: A simple wrapper for a generic array of spatial poses. This is the decoupled spatial/temporal context for a hierarchy: it represents the spatial description of a hierarchy at some point in time (i.e. the sample data in a keyframe for a hierarchy).
        - *Spatial poses*: A pointer to the start of the individual node pose array (data actually owned by the next structure).
        - *Constructor/factory/initialize*: Assign default values (e.g. call the initializer for each spatial pose).
    - ***Hierarchical pose pool***: A pool or group of hierarchical poses and their individual node poses.
        - *Hierarchy*: A pointer or reference to the hierarchy associated with this spatial data. The pose pool describes the spatial properties of the hierarchy (where are they in space), while the hierarchy describes the overall organization of the poses (how are the poses related).
        - *Spatial pose pool*: The actual array of individual node poses. This is comparable to the 'keyframe pool' for all poses associated with a hierarchy and its nodes. To clarify: this is the set of all poses for all nodes.
        - *Hierarchical poses*: An array of hierarchical poses (referencing the spatial poses). This is comparable to the 'keyframe pool' for a whole hierarchy. This is what organizes the above individual node poses.
        - *Channels*: An array of transformation channels for each node in the hierarchy; describes which individual pose transformation components are used by each node (e.g. rotation x, translation xyz, etc.); this is useful for optimization later.
        - *Euler order*: Some global flag for the pool that describes the concatenation order of orientation channels.
        - *Hierarchical pose count*: Number of hierarchical poses.
        - *Spatial pose count*: Total number of spatial poses: hierarchical pose count times

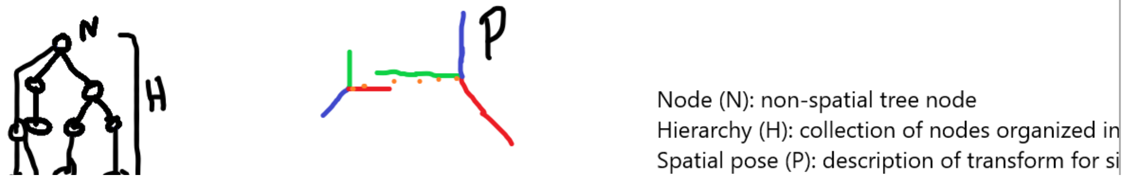
        hierarchy node count.
    - *Constructor/factory/initialize*: Given the associated hierarchy, allocate the full list of spatial poses, hierarchical poses and channels, wiring them up and assigning default values (e.g. call the initializer for each hierarchical pose, or each spatial pose).
    - *Destructor/release/cleanup*: Deallocate the data.

3. ***Kinematics***: Here we implement the process for representing our hierarchical data in the common space at any given time: forward kinematics.
    - ***Hierarchical state***: Much like our clip controller interface is responsible for managing clips, we also have an interface for controlling hierarchical pose data. The *hierarchical state* describes the complete pose information for a given hierarchy. The objective of this interface is to aggregate all of the information about a hierarchical pose and convert it to a format that can be used by other systems, such as graphics, at any time. The final result is a set of transformations relative to the root node's parent space (i.e. object-space; e.g. the object could be a "skeleton").
        - *Hierarchy*: A pointer or reference to the hierarchy associated with this spatial state.
        - *Sample pose*: A hierarchical pose representing each node's animated pose at the current time.
        - *Local-space pose*: A hierarchical pose representing each node's transformation relative to its parent's space.
        - *Object-space pose*: A hierarchical pose representing each node's transformation relative to the root's parent space (the actual object that the hierarchy represents).
        - *Constructor/factory/initialize*: Given the associated hierarchy, allocate and reset the poses above.
        - *Destructor/release/cleanup*: Deallocate poses.
    - ***Forward kinematics***: The final piece to this setup is the *forward kinematics* ("FK") algorithm. Implement the algorithm with a hierarchical state as input:
        - For each node in the associated hierarchy:
            - If the node is *not* a root node, the node's object-space pose transform is the product of its parent's object-space pose transform, and its own local-space pose transform.
            - Otherwise (root node), the node's object-space pose transform is its own local-space pose transform.
4. ***Hierarchical pose-to-pose animation***: Here we set up data for hierarchical pose-to-pose animation.

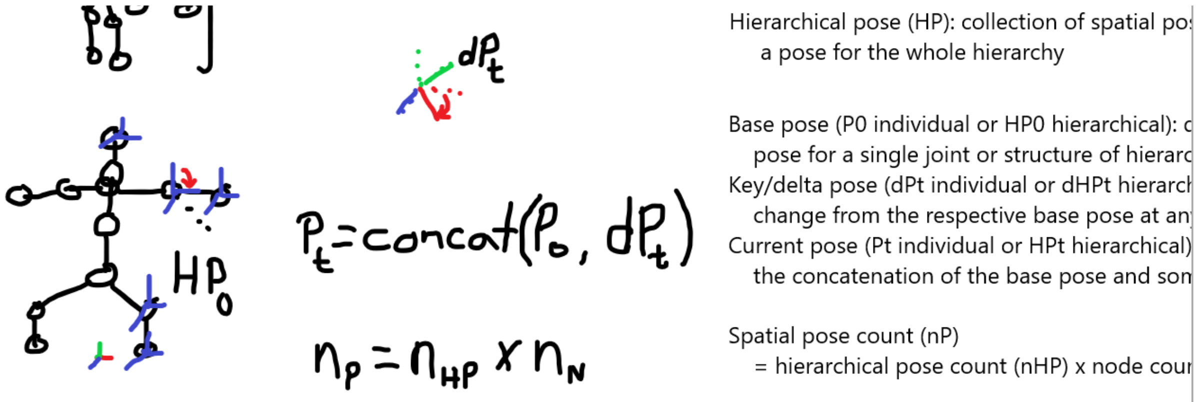
    - ***Base pose ("bind pose")***: The *base pose* defines the fixed default structure and figure of the hierarchy. In animation terms this is colloquially referred to as ***T-pose*** or ***A-pose*** because when a bipedal character holds this pose it stands in a T- or A-shaped posture (which we may refer to as "bind pose").
        - Create a hierarchy with enough nodes to represent a humanoid skeleton.

    - Initialize a hierarchical pose pool with 4 poses.
    - Initialize the first (index 0) as the base pose. This means that there must be one distinct spatial pose per node as this is ultimately responsible for setting the offsets between nodes (e.g. bones in a skeleton connect the joints).
    - See the demo in *animal3D* for an example of base pose data.
  - ***Key poses ("delta poses")***: The actual animation data for hierarchical poses is stored in *key poses* (which we may refer to as "delta poses"). These pose descriptions represent the change from the base pose, hence "delta". ***Pose-to-pose animation happens at this level***.
    - Implement 3 unique key poses, each modifying a subset of the nodes (e.g. the shoulders are revolute joints, so their key poses will change orientation; an entire skeleton can move if the root node's translation changes).
  - ***Calculate object-space pose***: The goal is to represent the nodes in a common *object-space* using forward kinematics. This process is broken down into four steps (referred to here as "phases" to differentiate from the 'step' interpolation function).
    - *Interpolation*: The first phase is to interpolate between key poses. For this assignment, implement the 'step' interpolation algorithm, which is effectively just *duplication* of a single input key pose. This phase takes key poses to be interpolated and stores the result in the target state's *sample pose* (for the 'step' function, this just means directly copying the pose from the pool into the state).
    - *Concatenation*: The next phase is to concatenate with the base pose. This phase takes the *base pose* from the source pool and the *sample pose* from the state as inputs, and *concatenates* them into the state's *local pose*.
    - *Conversion*: Since FK operates on matrices, we now convert our pose into a transformation matrix. The state's *local pose* is both the source and target, where the each raw pose description (orientation, scale, translation) gets converted into a matrix (transform).
    - *FK*: Finally, the forward kinematics algorithm is executed, converting the state's *local-space transforms* into *object-space transforms*.

  - Here is a diagram of it all:



Node (N): non-spatial tree node
Hierarchy (H): collection of nodes organized in
Spatial pose (P): description of transform for si



$$P_t = \text{concat}(P_0, dP_t)$$

$$n_P = n_{HP} \times n_N$$

Hierarchical pose (HP): collection of spatial po
a pose for the whole hierarchy

Base pose (P0 individual or HP0 hierarchical): c
pose for a single joint or structure of hierarc
Key/delta pose (dPt individual or dHPt hierarch
change from the respective base pose at an
Current pose (Pt individual or HPt hierarchical)
the concatenation of the base pose and son

Spatial pose count (nP)
= hierarchical pose count (nHP) x node cou

5. ***Basic testing interface***: Implement the following for your testing application:
   - ***Interface***: Set up the following in your testing application:
     - Instantiate and set up the hierarchy and poses described above. For context, the object we are testing is a "skeleton" and the hierarchy nodes represent "joints".
     - Instantiate and set up some hierarchy states (3+) to display the skeleton:
       - One state always displays the base pose.
       - At least one state allows the user to flip through all key poses (concatenated with base pose).
       - At least one state flips through the key poses automatically using a clip controller (one keyframe per pose, whose data are just indices, and a single clip to sequence them). This will behave as the 'step' interpolation function over time.
   - ***Input***: Add controls for the following tasks:
     - Select hierarchical state to edit.
     - Add controls to flip through the individual nodes of the currently edited state. Show the base pose for the current node and the current key pose ("identity" if we are showing the base pose).
     - Add buttons to flip through the key poses in the manual state.
     - For the clip controller state, use the same basic controls as lab 1.
   - ***Update***: Perform the following tasks in your update loop:
     - Apply input where applicable.
     - Update all controllers.
     - For each hierarchical state, run the 4-step update algorithm: interpolate, concatenate, convert, FK.
   - ***Display***: Display the following information:
     - Display the skeletal joints as small spheres. They should be positioned correctly in the global space to display an articulated figure. Consider drawing lines

       connecting each joint to its parent for better visibility.
     - Display all controllers and their info (all members) simultaneously as text (same as lab 1).
     - Display user controls for interaction and feedback.

Bonus:
You are encouraged to complete one or more of the following bonus opportunities (rewards listed):

- ***Inverse kinematics (+1)***: Implement the fundamental inverse kinematics algorithm. This is used to convert the global pose back to local.
    - *Object-space inverse pose*: Add another hierarchical pose to the hierarchical state interface to represent the inverse of the object-space pose.
    - Given a target hierarchical state as input, here is the algorithm:
        - For each node in the associated hierarchy:
            - If the node is *not* a root node, its local-space pose transform is the product of its parent's object-space *inverse* pose transform, and its own object-space pose transform.
            - Otherwise (root node), the node's local-space pose transform is its own object-space pose transform.
- ***Restore pose data (+1)***: While the conversion function gives us a transformation matrix from the raw pose description, the reverse of this function should restore the description from a transformation matrix. This can be used to display the raw data for each pose phase (base, key, sample, local, object) instead of just the original data (base, key).

Coding Standards:
You are required to mind the following standards (penalties listed):

- ***Reminder: You may be referencing others' ideas and borrowing their code. Credit sources and provide a links wherever code is borrowed, and credit your instructor for the starter framework, even if it is adapted, modified or reworked (failure to cite sources and claiming source materials as one's own results in an instant final grade of F in the course)***. Recall that borrowed material, even when cited, is not your own and will not be counted for grades; therefore you must ensure that your assignment includes some of your own contributions and substantial modification from what is provided. ***This principle applies to all evaluations***.
- ***Reminder: You must use version control consistently (zero on organization)***. Commit after a small change set (e.g. completing a section in the book) and push to your repository once in a while. Use branches to separate features (e.g. a chapter in the book), merging back to the parent branch (dev) when you stabilize something.
- ***Visual programming interfaces (e.g. Blueprint) are forbidden (zero on assignment)***. The programming languages allowed are: C, C++ and/or C#.
    - If you are using Unity, all front-end code must be implemented in C# (i.e. without the use of additional editors). You may implement and use your own C/C++ back-end plugin. The editor may be used strictly for UI (not the required algorithms).
    - If you are using Unreal, all code must be implemented directly in C/C++ (i.e. without Blueprint). Blueprints may be used strictly for UI (not the required algorithms).
    - You have been provided with a C-based framework called *animal3D* from your instructor.

  - You may find another C/C++ based framework to use. Ask before using.
- ***The 'auto' keyword and other language equivalents are forbidden (-1 per instance)***. Determine and use the proper variable type of all objects. Be explicit and understand what your data represents. Example:
  - auto someNumber = 1.0f;
    - This is a float, so the correct line should be: float someFloat = 1.0f;
  - auto someListThing = vector<int>();
    - You already know from the constructor that it is a vector of integers; name it as such: vector<int> someVecOfInts = vector<int>();
  - Pro tip: If you don't like the vector syntax, use your own typedefs. Here's one to make the previous example more convenient:
    - typedef vector<int> vecInt;
    - vecInt someVecOfInts = vecInt();
- ***The 'for-each' loop syntax is forbidden (-1 per instance)***. Replace 'for each' loops with traditional 'for' loops: the loops provided have this syntax:
  - In C++: for (<object> : <set>)...
  - In C#: foreach (<object> in <set>)...
- ***Compiler warnings are forbidden (-1 per instance)***. Your starter project has warnings treated as errors so you must fix them in order to complete a build. ***Do not disable this***. Fix any silly errors or warnings for a nice, clean build. They are generally pretty clear but if you are confused please ask for help. Also be sure to test your work and product before submitting to ensure no warnings/errors made it through. This also applies to C# projects.
- ***Every section/block of code must be commented (-1 per ambiguous section/block)***. Clearly state the intent, the 'why' behind each section/block. This is to demonstrate that you can relate what you are doing to the subject matter.
- ***Add author information to the top of each code file (-1 for each omission)***. If you have a license, include the boiler plate template (fill it in with your own info) and add a separate block with: 1) the name and purpose of the file; and 2) a list of contributors and what they did.

Points	5
Submitting	a text entry box or a website url

Due	For	Available from	Until
-	Everyone	-	-

GraphicsAnimation-Master-Range

Criteria	Ratings			Pts
IMPLEMENTATION: Architecture & Design Practical knowledge of C/C++/API/framework programming, engineering and architecture within the provided framework or engine.	**1 to >0.5 pts** **Full points** Strong evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely both efficient and functional.	**0.5 to >0.0 pts** **Half points** Mild evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely either efficient or functional.	**0 pts** **Zero points** Weak evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely neither efficient nor functional.	1 pts
IMPLEMENTATION: Content & Material Practical knowledge of content relevant to the discipline and course (e.g. shaders and effects for graphics, animation algorithms and techniques, etc.).	**1 to >0.5 pts** **Full points** Strong evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely both efficient and functional.	**0.5 to >0.0 pts** **Half points** Mild evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely either efficient or functional.	**0 pts** **Zero points** Weak evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely neither efficient nor functional.	1 pts
DEMONSTRATION: Presentation & Walkthrough Live presentation and walkthrough of code, implementation, contributions, etc.	**1 to >0.5 pts** **Full points** Strong evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely both accurate and confident.	**0.5 to >0.0 pts** **Half points** Mild evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely either accurate or confident.	**0 pts** **Zero points** Weak evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely neither accurate nor confident.	1 pts
DEMONSTRATION: Product & Output Live showing and explanation of final working implementation, product and/or outputs.	**1 to >0.5 pts** **Full points** Strong evidence of correct and stable final product that runs as expected; end result is largely both correct and stable.	**0.5 to >0.0 pts** **Half points** Mild evidence of correct and stable final product that runs as expected; end result is largely either correct or stable.	**0 pts** **Zero points** Weak evidence of correct and stable final product that runs as expected; end result is largely neither correct nor stable.	1 pts

Criteria	Ratings			Pts
ORGANIZATION: Documentation & Management Overall developer communication practices, such as thorough documentation and use of version control.	**1 to >0.5 pts** **Full points** Strong evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely both documented and organized.	**0.5 to >0.0 pts** **Half points** Mild evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely either documented or organized.	**0 pts** **Zero points** Weak evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely neither documented nor organized.	1 pts
BONUSES Bonus points may be awarded for extra credit contributions.	**0 pts** **Points awarded** If score is positive, points were awarded for extra credit contributions (see comments).		**0 pts** **Zero points**	0 pts
PENALTIES Penalty points may be deducted for coding standard violations.	**0 pts** **Points deducted** If score is negative, points were deducted for coding standard violations (see comments).		**0 pts** **Zero points**	0 pts
				Total Points: 5