Lab 3: Pose Blending Operations

Published | Edit | ⋮

GPR-450 Advanced Animation Programming
Instructor: Daniel S. Buckstein
Lab 3: Pose Blending Operations

Summary:
In this lab we dive deeper into pose-to-pose skeletal animation. Using the data structures and algorithms described in lab 2 and project 2, we add the functionality of many more blending operations. This lab focuses on the algorithms used to design a variety of pose-to-pose animation. This lab is designed to prepare you for project 3, in which we design data structures and systems around these operations.

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:
This assignment focuses on fundamental blending algorithms and requires abstract thinking. You will apply what you traditionally know as spatial algorithms (e.g. waypoints) to entire spatial poses (TRS transforms) and hierarchical poses (skeletons).

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, Maya and Python), complete the following steps:

1. ***Repository setup***: See lab 1 for the complete setup instructions.
   - Check out the '*anim/blend*' branch from the course repository, it has some stuff you can start with.
   - *Hint: We have already partially implemented some of the stuff below, so look for uses of it in prior commits in the code base, your previous assignments and our in-class work.*
2. ***Operation interface***: Implement **one class or interface to contain all of the code described in parts 3-4**. This interface should reference your data structures for *spatial pose* and *hierarchical pose*, as described in the previous lab and project.
   - A *blend operation* is a function that behaves much like a mathematical operator for *spatial poses*. In the same fashion as mathematical operators and functions, they may have zero or more parameters and may either return a new spatial pose, and/or set/modify one passed in as the first parameter (see *animal3D*'s vector math functions for an example of this style).
   - Operations should always return a new pose or a pointer/reference to a spatial pose, even if there is a 'return' parameter, to allow operation chaining (*animal3D*'s vector functions do this).
   - The specifications for each operation are described in parts 3-4, but in general, the parameters may consist of:

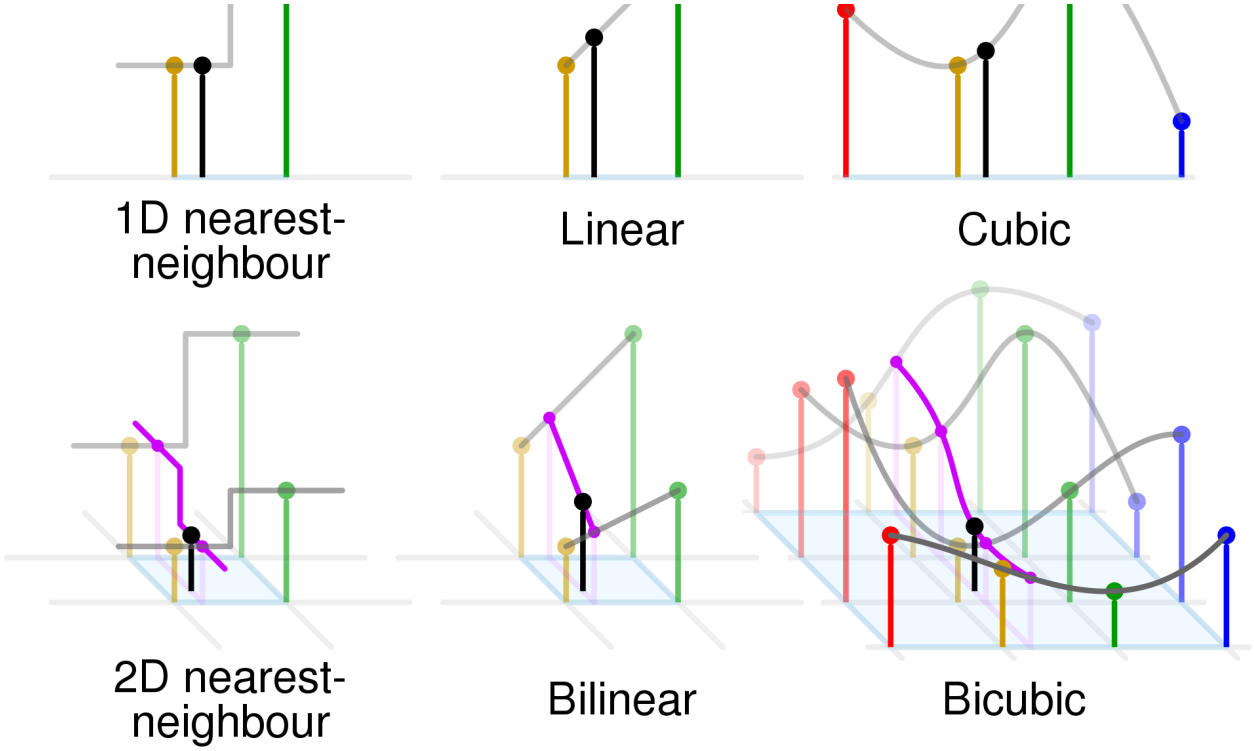
    - Zero or more pointers or references to *spatial poses* (optionally, one additional parameter for capturing the result) and/or other representations of pose data. These are described in each operation below as *controls*. In plain math, these are the subscripts of a function name describing *mathematical constants*.
    - Zero or more *independent variables* or *input parameters*. These are described below as *inputs*. These are the mathematical inputs to the function (in the parentheses) that are intended to change with respect to time, therefore establishing them as the *independent variables* related to the functions. Generally they are some representation of time, but there are times where they represent something else, or are excluded entirely. Generally, the operators should be differentiable with respect to these inputs, which is what distinguishes them from constants.
  - In the specifications below, functions that directly map to a C++ operator or constructor may be implemented as such for convenience; these are marked with an asterisk (*). Consider implementing the operator itself (e.g. 'operator +') and the corresponding assignment operator (e.g. 'operator +=').
3. ***Fundamental blend operations***: Implement the following fundamental blend operations. These are the core operations that drive pose-to-pose animation. While many of them are standard operations used in many engines, some of them are creative yet useful for future implementations (and your instructor thinks they are cool). The names are not definitive of what they do; mind the intent of each operation as you implement them and customize as needed.
  - **Identity**: Always returns/sets the constant identity pose.
    - *Formats:* $\mathrm{identity}(\,)$; $\mathrm{id}(\,)$.
    - *Return*: identity pose.
  - **Construct***: Equivalent to a constructor, this operation returns/sets a pose constructed using the components provided.
    - *Formats*: $\mathrm{construct}_{r,s,t}(\,)$; $\mathrm{pose}_{r,s,t}(\,)$.
    - *Return*: new pose with validated control values as components.
    - *Controls* (3): vectors representing rotation angles, scale and translation.
  - **Constant/copy***: Equivalent to unary plus/positive (constant) or the assignment operator (copy), this operation simply returns/sets the unchanged control pose. *Note: These can be the same operation or broken into two, depending on the language (C/C++) and/or how you arrange the function parameters*.
    - *Formats:* $\mathrm{constant}_P(\,)$; $\mathrm{copy}_P(\,)$; $\mathrm{plus}_P(\,)$.
    - *Return*: control pose.
    - *Controls* (1): spatial pose.
  - **Negate/invert***: Equivalent to unary minus/negative, this operation calculates the opposite/inverse pose description that "undoes" the control pose. Note that this may not be a literal negation as each component may follow different rules for inversion.
    - *Formats:* $\mathrm{negate}_P(\,)$; $\mathrm{invert}_P(\,)$; $\mathrm{minus}_P(\,)$.
    - *Return*: inverted/negated control pose.

    - *Controls* (1): spatial pose.
  - **Concatenate/merge***: Similar to binary plus/addition, this operation calculates the "sum" or "merging" of the two control poses. It is effectively a piecewise concatenation as each component of a spatial pose follows different rules for concatenation; note that it may not be a commutative or associative operation.
    - *Formats:* $\mathrm{concat}_{P_{lh},P_{rh}}(\,)$; $\mathrm{add}_{P_{lh},P_{rh}}(\,)$; $\mathrm{merge}_{P_{lh},P_{rh}}(\,)$.
    - *Return*: concatenation of control poses.
    - *Controls* (2): left-hand and right-hand spatial poses.
  - **Nearest**: Selects one of the two control poses using nearest interpolation.
    - *Formats:* $\mathrm{nearest}_{P_0,P_1}(u)$; $\mathrm{near}_{P_0,P_1}(u)$.
    - *Return*: nearest pose given input.
    - *Controls* (2): initial and terminal spatial poses.
    - *Inputs* (1): blend parameter; an input of less than 0.5 results in the initial/first control pose (P0); an input of 0.5 or greater results in the terminal/second control pose (P1).
  - **Linear interpolate/blend/mix**: Implements a linear interpolation algorithm for poses (i.e. lerp). Note that pose components may have different rules for interpolation.
    - *Formats:* $\mathrm{lerp}_{P_0,P_1}(u)$; $\mathrm{blend}_{P_0,P_1}(u)$; $\mathrm{mix}_{P_0,P_1}(u)$.
    - *Return*: linear blend between control poses.
    - *Controls* (2): initial and terminal spatial poses.
    - *Inputs* (1): blend parameter; an input of 0 results in P0; an input of 1 results in P1; any other input between 0 and 1 results in a linear blend or mixture of the two control poses.
  - **Cubic**: Implements a cubic interpolation algorithm for poses (e.g. Catmull-Rom).
    - *Formats:* $\mathrm{cubic}_{P_{-1},P_0,P_1,P_2}(u)$.
    - *Return*: cubic blend between initial and terminal control poses (P0 and P1).
    - *Controls* (4): pre-initial, initial, terminal and post-terminal poses.
    - *Inputs* (1): blend parameter; an input of 0 results in P0; an input of 1 results in P1; any other input between 0 and 1 results in a cubic blend or mixture of the two control poses.
4. ***Derivative blend operations***: Implement the following derivative blend operations. They are based on, or they build upon, the same principles of those in part 3, however they are a bit more complex.
  - **Deconcatenate/split***: Similar to binary minus/subtraction, this operation calculates the "difference" or "split" between the two control poses. The same rules of concatenation apply.
    - *Formats:* $\mathrm{deconcat}_{P_{lh},P_{rh}}(\,)$; $\mathrm{sub}_{P_{lh},P_{rh}}(\,)$; $\mathrm{split}_{P_{lh},P_{rh}}(\,)$.
    - *Return*: deconcatenation of control poses.
    - *Controls* (2): left-hand and right-hand spatial poses.
  - **Scale***: Similar to scalar multiplication, this operation calculates the "scaled" pose, which is some blend between the identity pose and the control pose.
    - *Formats:* $\mathrm{scale}_P(u)$; $\mathrm{mul}_P(u)$.

- *Return*: blend between identity and control pose.
- *Controls* (1): spatial pose.
- *Inputs* (1): blend parameter; an input of 0 results in the identity pose; an input of 1 results in the control pose; any other input between 0 and 1 results in some pose that is not identity but not quite the control pose.

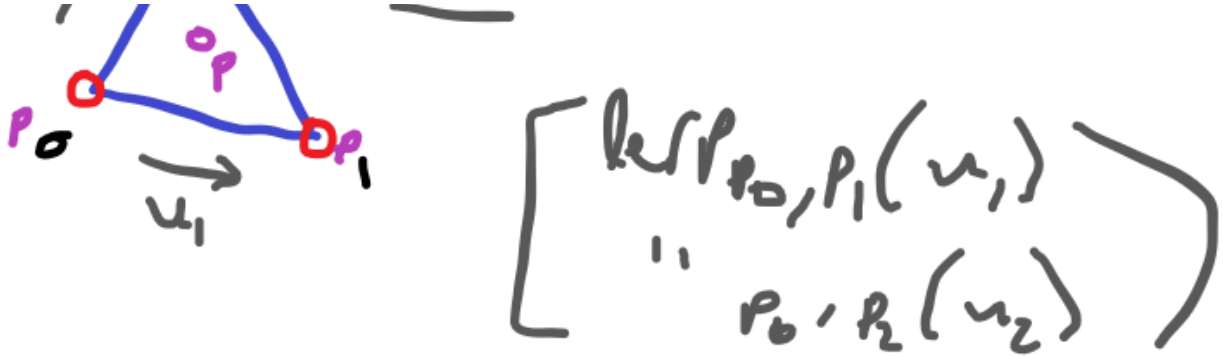
- **Triangular**: Implements triangular interpolation for poses.
  - *Formats:* $\text{triangular}_{P_0,P_1,P_2}(u_1,u_2)$; $\text{tri}_{P_0,P_1,P_2}(u_1,u_2)$.
  - *Return*: blend between three control poses.
  - *Controls* (3): three spatial poses.
  - *Inputs* (2): scaling parameters corresponding to the respective target poses; internal calculation of a third parameter, $u_0 = 1 - u_1 - u_2$, is required to scale the initial pose (P0).
- **Bi-nearest**: Implements the bilinear interpolation function for poses.
  - *Formats:* $\text{binearest}_{P_{0,0},P_{0,1},P_{1,0},P_{1,1}}(u_0,u_1,u)$.
  - *Return*: nearest neighbor of two other nearest neighbor calculations.
  - *Controls* (4): two pairs of initial and terminal spatial poses.
  - *Inputs* (3): blend parameters, each corresponding to a pair of poses: the first is used to select the nearest pose in the first pair of control poses, then the next pair, finally performing the nearest operation on the results of each pair.
- **Bi-linear**: Implements the bilinear interpolation function for poses.
  - *Formats:* $\text{bilerp}_{P_{0,0},P_{0,1},P_{1,0},P_{1,1}}(u_0,u_1,u)$.
  - *Return*: bilinear blend between control poses.
  - *Controls* (4): two pairs of initial and terminal spatial poses.
  - *Inputs* (3): blend parameters, each corresponding to a pair of poses: the first is used to blend the first pair of control poses, then the next pair, finally blending the results of each blended pair.
- **Bi-cubic**: Implements a bicubic interpolation algorithm for poses.
  - *Formats:* $\text{bicubic}_{P_{-1,-1}\ldots P_{2,2}}(u_{-1},u_0,u_1,u_2,u)$.
  - *Return*: bicubic blend between control poses.
  - *Controls* (16): four sets of cubic control poses.
  - *Inputs* (5): blend parameters, each corresponding to a set of cubic poses: behaves similarly to bilinear, but the final result is the cubic blend of other cubic blends.
- Here are the slides about **blend operations** ⬇ **(https://champlain.instructure.com/courses/908328/files/193719045/download?download_frd=1)** .

- Here is a diagram of some of the interpolation algorithms. Think of each of the dots more abstractly: they are no longer just points in space, they now represent complete poses.



By Cmglee - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=53064904
(https://commons.wikimedia.org/w/index.php?curid=53064904)

- Triangular:



$$P = u_0 P_0 \oplus u_1 P_1 \oplus u_2 P_2$$

```
P = concat(concat(scale(P0, u0), scale(P1, u1)), scale(P2
```

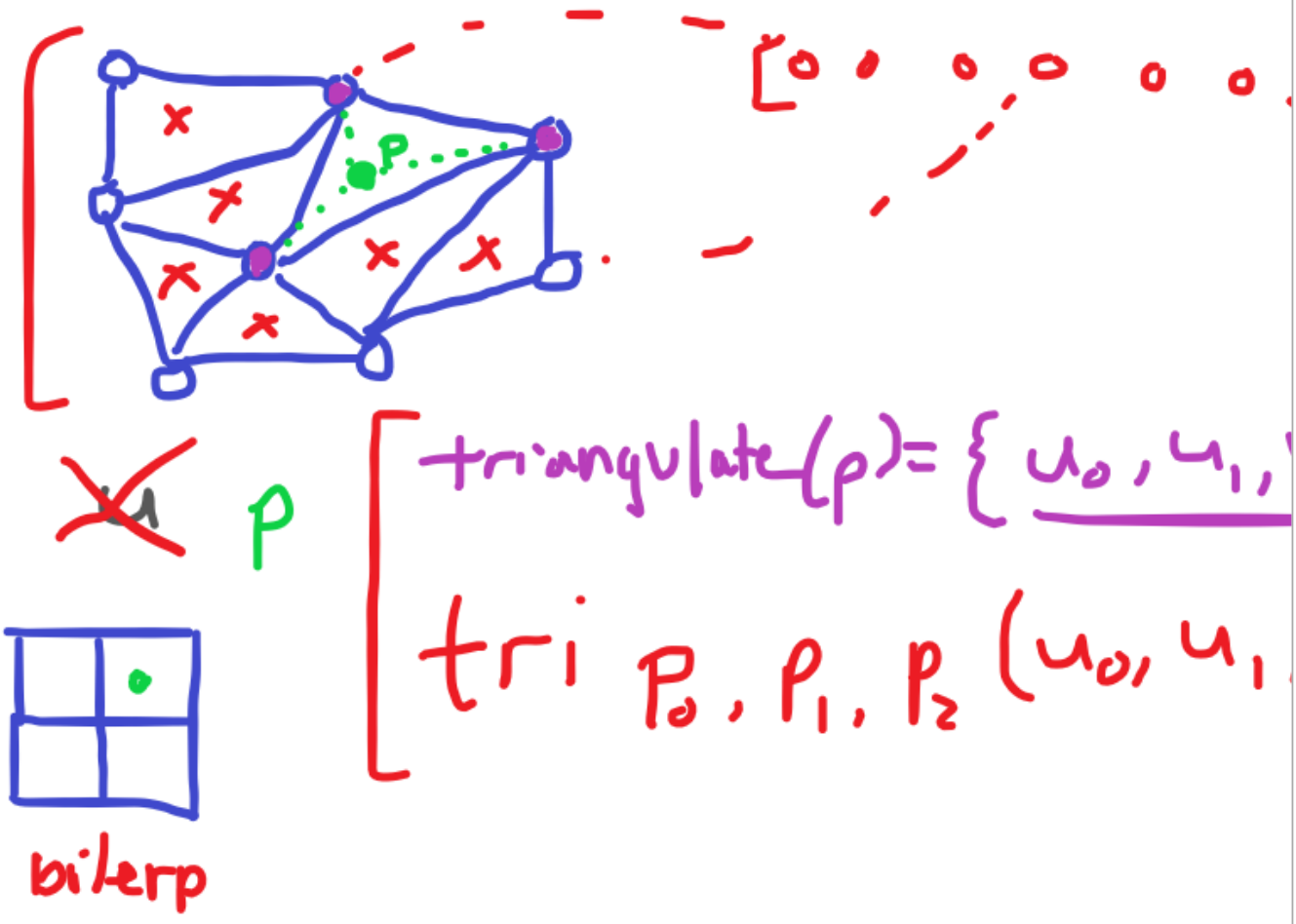
5. ***Hierarchical blend operations***: Implement each of the operations in parts 3-4 as *hierarchical pose* operations.
   - A *hierarchical blend operation* has the same format as the spatial pose operations, specified in part 2, but the parameter type changes to a *hierarchical pose*.
   - This is easier than it sounds: since a hierarchical pose is just a set of spatial poses (one per node), each hierarchical blend operation just loops through the nodes, calling the respective *spatial pose* operation for each node! That said, the only additional *control* parameter for each operation is the number of nodes.
6. ***Basic testing interface***: Implement the following for your testing application:
   - ***Interface***: Set up the following in your testing application:
     - Instantiate a clip controller or a simple timer to maintain and update the current keyframe time.
     - Instantiate and set up one hierarchy, including its base pose data and a minimum of 4 distinct key poses. The key poses are controlled by the clip controller, playing on loop.
       - Set up the skeleton and pose data either manually (e.g. in earlier course repository states) or using a loader (e.g. from project 2).
     - Instantiate and set up a minimum of 4 hierarchical states:
       - ***Base pose***: The first state is persistent and completely initialized *once, on-load,* to represent the base pose.
       - ***Output***: The second state will always maintain the output state, or result, of your currently-tested blend operation.
       - ***Controls***: The remaining states will always maintain the control states for your currently-tested blend operation. The number of *active* controls will vary based on the current operation.

- ***Input***: Add controls for the following tasks:
  - Allow the user to toggle which blend operation to test.
  - The clip controller uses only the most basic controls (play forward/reverse, pause).
- ***Update***: Perform the following tasks in your update loop:
  - Apply input where applicable.
  - Update all controllers and states.
  - For each hierarchical state, run the 4-step update algorithm: interpolate, concatenate, convert, FK. Due to the nature of this assignment, one major difference from the previous is that the first two stages may now be achieved using the very operations you have created!
- ***Display***: Display the following information:
  - Display a *complete skeleton* for the the *output pose* of the currently-tested operation, and *each active control pose*.
    - E.g. the '*scale*' operation is unary, so you should see only the scaled pose and the original input pose.
    - E.g. the '*lerp*' operation is binary, so you should see the interpolated pose and the two input poses.
  - Display any *independent variables* or *input parameters* used for the current operation.
    - E.g. the '*invert*' and '*concat*' operations do not have any independent variables (each input pose is considered a mathematical constant).
    - E.g. the '*lerp*' and '*scale*' operations each have one input parameter.
    - E.g. the '*Delaunay*' operation (see bonus) uses a 2D point.
  - Displaying the base pose state is optional as the base pose may be used as one of the control poses.
  - 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):

- ***Composite operations (+1)***: Write a second version of *lerp* that deconstructs the existing *lerp* operation into more primitive operations, representing the core mathematical formula. Compare the results of the two *lerp* operation versions (default vs composite). Here are two versions of lerp, using basic operations:
  - $\mathrm{LERP}_{P_0,P_1}(u) = P_0 + (P_1 - P_0)u$
  - $\mathrm{LERP}_{P_0,P_1}(u) = (1-u)P_0 + (u)P_1$
- ***Delaunay blending (+2)***: Implement a blend operation using Delaunay triangulation. This is reflective of the backend of some rigging tool that a technical artist might implement.
  - Each input pose is represented by a 2D point in some set of points (e.g. positioned on a

grid), arranged into triangles. A control point moves around the and is triangulated in that set, giving the two independent variables to the triangular blend operation.
    - *Format*: $\text{Delaunay}_{p[\,],P[\,]}(q)$.
    - *Return*: triangulated pose corresponding to control point.
    - *Controls* (2): grid/set of 2D points, arranged into triangles, each representing a spatial/hierarchical pose; a mapping of the 2D points to their respective poses.
    - *Inputs* (1): 2D point.
  - Diagram:



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++, C# (Unity) and/or Python (Maya).
  - 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).
  - If you are using Maya, all code must be implemented in Python. You may implement and use your own C/C++ back-end plugin. Editor tools may be used for UI.
  - 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