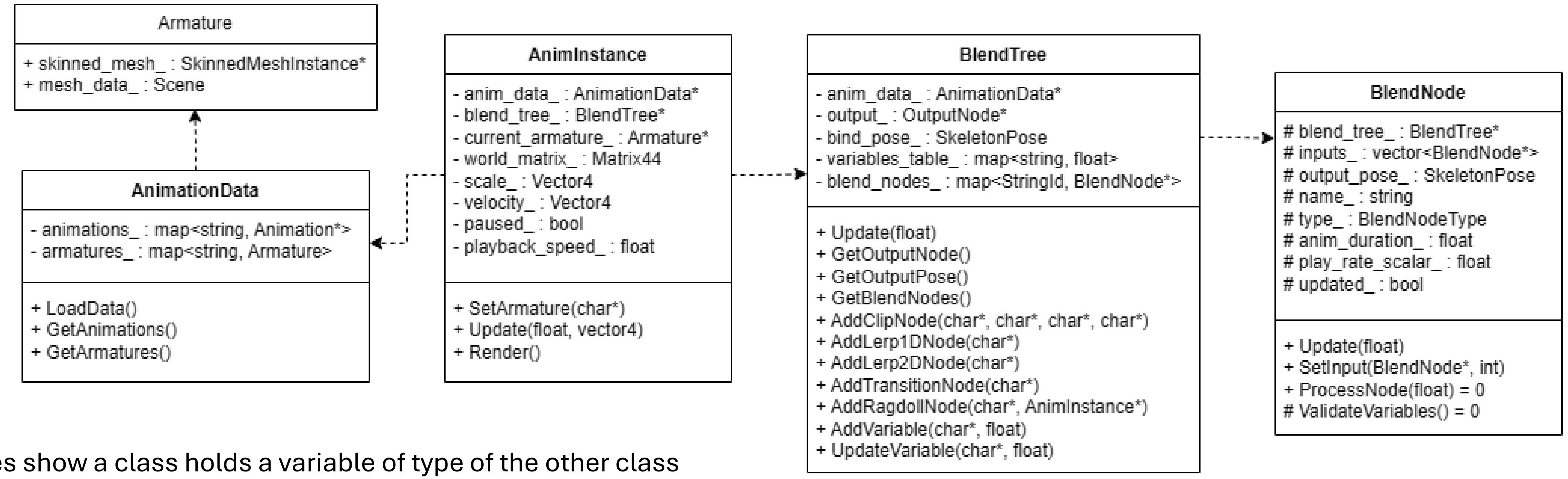
3D System Class Diagram

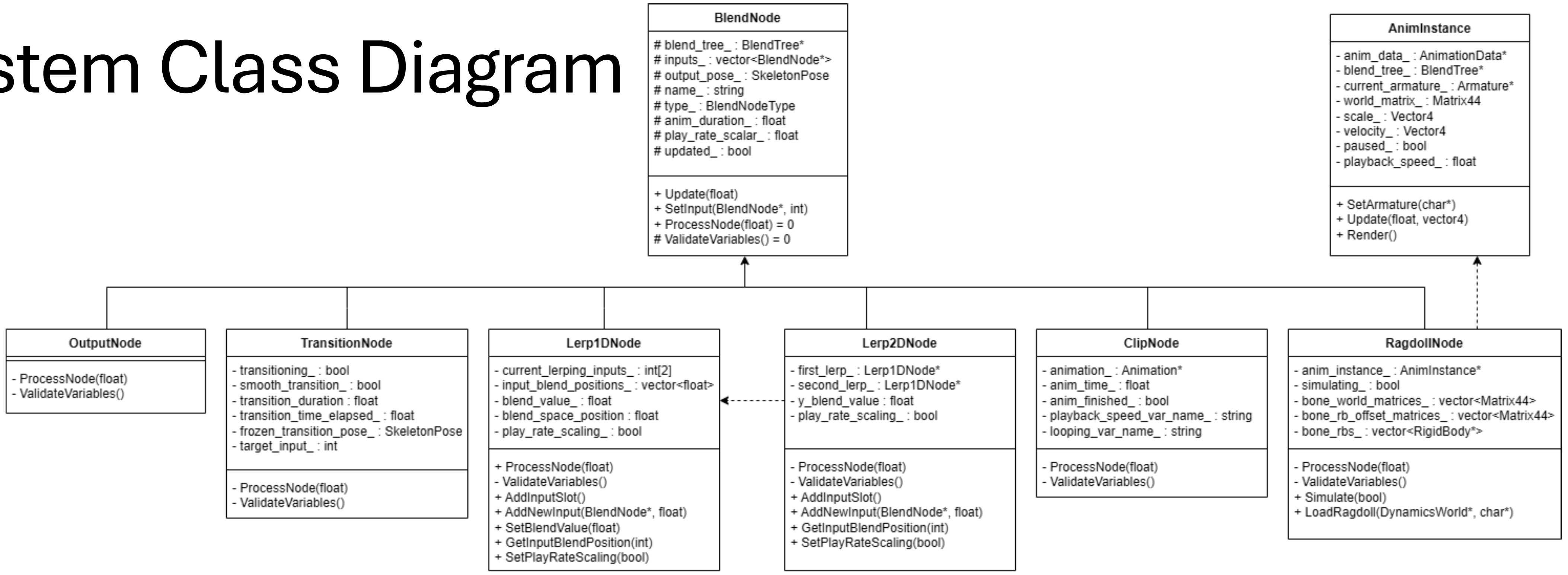

Armature
+ skinned_mesh_ : SkinnedMeshInstance*
+ mesh_data_ : Scene
AnimationData
- animations_ : map<string, Animation*>
- armatures_ : map<string, Armature>
+ LoadData()
+ GetAnimations()
+ GetArmatures()
AnimInstance
- anim_data_ : AnimationData*
- blend_tree_ : BlendTree*
- current_armature_ : Armature*
- world_matrix_ : Matrix44
- scale_ : Vector4
- velocity_ : Vector4
- paused_ : bool
- playback_speed_ : float
+ SetArmature(char*)
+ Update(float, vector4)
+ Render()
BlendTree
- anim_data_ : AnimationData*
- output_ : OutputNode*
- bind_pose_ : SkeletonPose
- variables_table_ : map<string, float>
- blend_nodes_ : map<StringId, BlendNode*>
+ Update(float)
+ GetOutputNode()
+ GetOutputPose()
+ GetBlendNodes()
+ AddClipNode(char*, char*, char*, char*)
+ AddLerp1DNode(char*)
+ AddLerp2DNode(char*)
+ AddTransitionNode(char*)
+ AddRagdollNode(char*, AnimInstance*)
+ AddVariable(char*, float)
+ UpdateVariable(char*, float)
BlendNode
# blend_tree_ : BlendTree*
# inputs_ : vector<BlendNode*>
# output_pose_ : SkeletonPose
# name_ : string
# type_ : BlendNodeType
# anim_duration_ : float
# play_rate_scalar_ : float
# updated_ : bool
+ Update(float)
+ SetInput(BlendNode*, int)
+ ProcessNode(float) = 0
# ValidateVariables() = 0

Dotted lines show a class holds a variable of type of the other class
Many getters and setters have been omitted, along with certain variables – I've included what I consider most important

An application will have one AnimationData object, responsible for loading and storing all animation and mesh/skeletal data.
An armature is just what skinned mesh we are using – holds mesh and skeletal information.
Different animating game objects may need to play different animations, or the same animation but not in sync. So AnimInstance is a unique instance of animation playback.
Each AnimInstance has its own BlendTree (more UML next slide). The application has gui controlling one blend tree for one instance.
Animations and armatures are loaded once and then referenced by animation instances, rather than each instance loading and storing its own copy.

An instance may use any armature and any animation, it's up to the user to ensure they make sense. Game Engine Architecture by Jason Gregory (2019) describes this sort of decoupling.

3D System Class Diagram

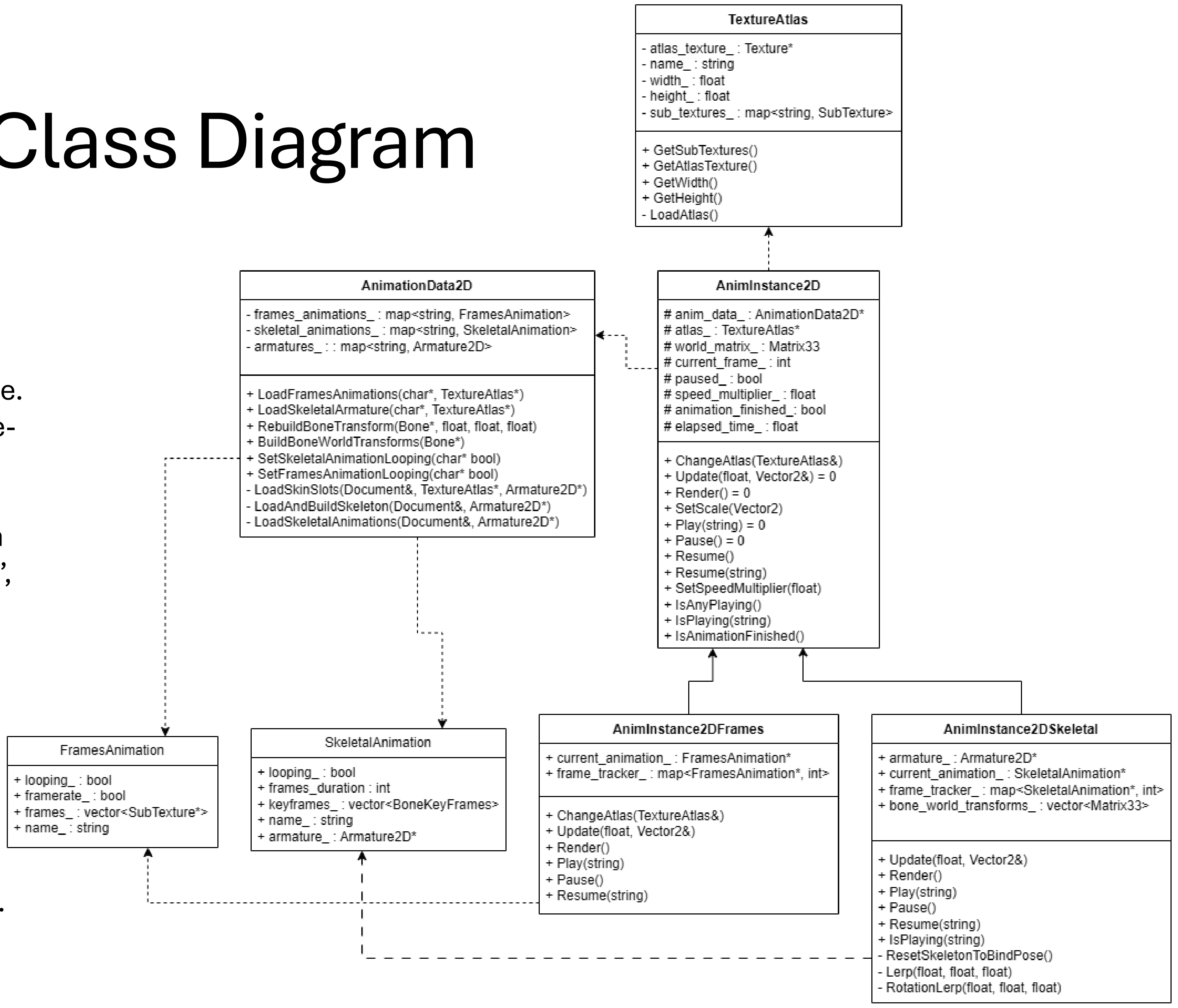

BlendNode
# blend_tree_ : BlendTree*
# inputs_ : vector<BlendNode*>
# output_pose_ : SkeletonPose
# name_ : string
# type_ : BlendNodeType
# anim_duration_ : float
# play_rate_scalar_ : float
# updated_ : bool
+ Update(float)
+ SetInput(BlendNode*, int)
+ ProcessNode(float) = 0
# ValidateVariables() = 0
AnimInstance
- anim_data_ : AnimationData*
- blend_tree_ : BlendTree*
- current_armature_ : Armature*
- world_matrix_ : Matrix44
- scale_ : Vector4
- velocity_ : Vector4
- paused_ : bool
- playback_speed_ : float
+ SetArmature(char*)
+ Update(float, vector4)
+ Render()
OutputNode
- ProcessNode(float)
- ValidateVariables()
TransitionNode
- transitioning_ : bool
- smooth_transition_ : bool
- transition_duration : float
- transition_time_elapsed_ : float
- frozen_transition_pose_ : SkeletonPose
- target_input_ : int
- ProcessNode(float)
- ValidateVariables()
Lerp1DNode
- current_lerping_inputs_ : int[2]
- input_blend_positions_ : vector<float>
- blend_value_ : float
- blend_space_position : float
- play_rate_scaling_ : bool
+ ProcessNode(float)
- ValidateVariables()
+ AddInputSlot()
+ AddNewInput(BlendNode*, float)
+ SetBlendValue(float)
+ GetInputBlendPosition(int)
+ SetPlayRateScaling(bool)
Lerp2DNode
- first_lerp_ : Lerp1DNode*
- second_lerp_ : Lerp1DNode*
- y_blend_value : float
- play_rate_scaling_ : bool
- ProcessNode(float)
- ValidateVariables()
+ AddInputSlot()
+ AddNewInput(BlendNode*, float)
+ GetInputBlendPosition(int)
+ SetPlayRateScaling(bool)
ClipNode
- animation_ : Animation*
- anim_time_ : float
- anim_finished_ : bool
- playback_speed_var_name_ : string
- looping_var_name_ : string
- ProcessNode(float)
- ValidateVariables()
RagdollNode
- anim_instance_ : AnimInstance*
- simulating_ : bool
- bone_world_matrices_ : vector<Matrix44>
- bone_rb_offset_matrices_ : vector<Matrix44>
- bone_rbs_ : vector<RigidBody*>
- ProcessNode(float)
- ValidateVariables()
+ Simulate(bool)
+ LoadRagdoll(DynamicsWorld*, char*)

Here’s the UML for the various blend nodes. Solid line shows inheritance. Variables with # are protected and therefore inherited.
The blend tree is a tree structure of blend nodes. There is one output node, which is the root. Update() acts recursively, calling Update() on all input nodes before validating necessary variables and finally processing the node.
Each node outputs a skeletal pose which may subsequently be used by parent nodes.
The Lerp2D node contains two internal Lerp1D nodes (to the user these are called collections), which are processed by the Lerp2D node itself, which is why Lerp1D node’s ProcessNode() is public.
RagdollNode needs a reference to the instance to get the instance’s translation (for updating rigid bodies). This brings the limitation that blend trees utilising ragdoll nodes cannot be shared among animation instances (which may sometimes be desired).

2D System Class Diagram

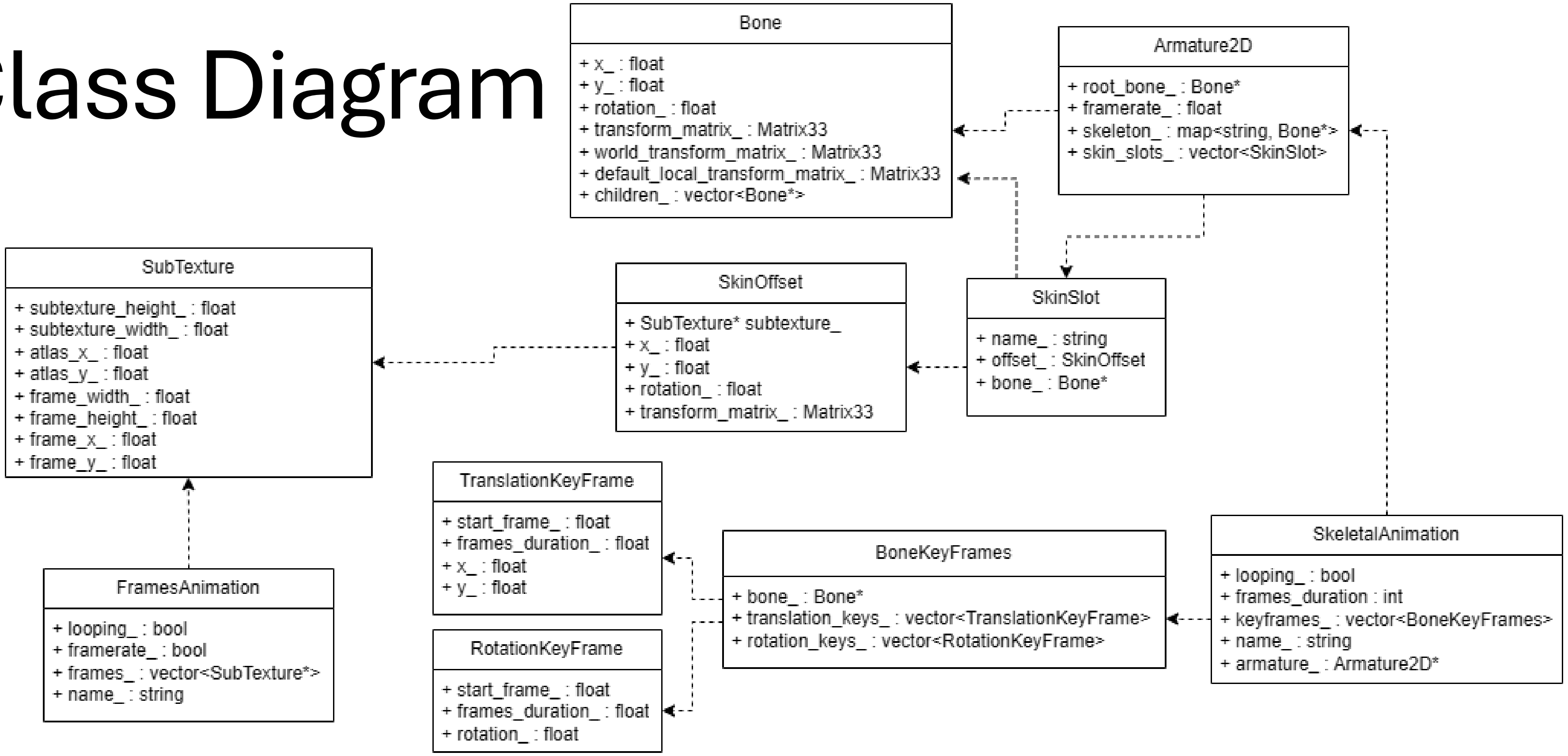
Similar to the 3D system, there's the same concept of the animation instance. There are classes for skeletal and frame-based animation instances.
A texture atlas stores a texture and subtextures. Each subtexture is either a frame of animation or a part of the 'skin', mapped onto a bone in the skeleton (more on next slide).
As with the 3D system, an AnimationData class stores all animation and armature information.

Unlike the 3D system, Skeletal animations can only be used with the specific armature they were created for. With further development this could be decoupled.



TextureAtlas
- atlas_texture_ : Texture*
- name_ : string
- width_ : float
- height_ : float
- sub_textures_ : map<string, SubTexture>
+ GetSubTextures()
+ GetAtlasTexture()
+ GetWidth()
+ GetHeight()
- LoadAtlas()
AnimationData2D
- frames_animations_ : map<string, FramesAnimation>
- skeletal_animations_ : map<string, SkeletalAnimation>
- armatures_ : : map<string, Armature2D>
+ LoadFramesAnimations(char*, TextureAtlas*)
+ LoadSkeletalArmature(char*, TextureAtlas*)
+ RebuildBoneTransform(Bone*, float, float, float)
+ BuildBoneWorldTransforms(Bone*)
+ SetSkeletalAnimationLooping(char* bool)
+ SetFramesAnimationLooping(char* bool)
- LoadSkinSlots(Document&, TextureAtlas*, Armature2D*)
- LoadAndBuildSkeleton(Document&, Armature2D*)
- LoadSkeletalAnimations(Document&, Armature2D*)
AnimInstance2D
# anim_data_ : AnimationData2D*
# atlas_ : TextureAtlas*
# world_matrix_ : Matrix33
# current_frame_ : int
# paused_ : bool
# speed_multiplier_ : float
# animation_finished_: bool
# elapsed_time_ : float
+ ChangeAtlas(TextureAtlas&)
+ Update(float, Vector2&) = 0
+ Render() = 0
+ SetScale(Vector2)
+ Play(string) = 0
+ Pause() = 0
+ Resume()
+ Resume(string)
+ SetSpeedMultiplier(float)
+ IsAnyPlaying()
+ IsPlaying(string)
+ IsAnimationFinished()
FramesAnimation
+ looping_ : bool
+ framerate_ : bool
+ frames_ : vector<SubTexture*>
+ name_ : string
SkeletalAnimation
+ looping_ : bool
+ frames_duration : int
+ keyframes_ : vector<BoneKeyFrames>
+ name_ : string
+ armature_ : Armature2D*
AnimInstance2DFrames
+ current_animation_ : FramesAnimation*
+ frame_tracker_ : map<FramesAnimation*, int>
+ ChangeAtlas(TextureAtlas&)
+ Update(float, Vector2&)
+ Render()
+ Play(string)
+ Pause()
+ Resume(string)
AnimInstance2DSkeletal
+ armature_ : Armature2D*
+ current_animation_ : SkeletalAnimation*
+ frame_tracker_ : map<SkeletalAnimation*, int>
+ bone_world_transforms_ : vector<Matrix33>
+ Update(float, Vector2&)
+ Render()
+ Play(string)
+ Pause()
+ Resume(string)
+ IsPlaying(string)
- ResetSkeletonToBindPose()
- Lerp(float, float, float)
- RotationLerp(float, float, float)

2D System Class Diagram



Frames animations are simply a vector of subtextures along with some simple metadata. The sprite simply updates what subtexture is being rendered each frame.
Skeletal animations require more info – they are a vector of bone keyframes, which contain translation and rotation information for the necessary bones.
When updating, each bone in the bone keyframes vector has its local transform recalculated based on the keyframe info.
The skeleton is a tree structure, so bones contain pointers to child bones. Once local transforms are updated, the tree is recursively traversed and new world transforms calculated.
The armature also contains all the bones of the skeleton, along with a vector of skin slots, which map skin offsets to the bones.

Analysis

- The systems are largely object oriented.
  - 3D blend tree: rapid development of new node types
  - 2D systems: different types of animation
  - Animation instances as objects

Analysis

- Minimized memory usage by storing as much data as possible only once.
  - Data shared among instances – not parallelisable.
  - Some redundant memory usage still unavoidable. E.g. each time a 2D skeletal instance updates, it modifies the armature it's referencing. Therefore it must then keep a local copy of the final world transforms so that other instances are free to then modify the same armature.

- Using pointers where possible
  - Cuts down on indirection of using indices for lookups

Analysis

- Further performance considerations:
  - The rigid bodies used in ragdoll simulation are set to sleep when the simulation is not ongoing. The ragdoll simulation leads to a drop in framerate, so only simulate when absolutely necessary.
  - With how the 2D blend node is currently implemented, the SetBlendValue() method is called multiple times on each internal blend node within the Lerp2D node when setting the blend values if play rate scaling is enabled.
    - With further development this should be tackled, especially by reducing the function into smaller functions each with a single purpose.