Crowd animation in Unity: implementation of a module utilizing Data-Oriented Technology Stack and a General-Purpose Graphics Processing Unit

Felipe Marques de Carvalho de Oliveira[1], **Ana Paula Canal**[1]

[1]Computer Science – Universidade Franciscana (UFN)
Mailbox 151 – 97.010-032 – Santa Maria – RS – Brazil

{felipe.carvalho, apc}@ufn.edu.br

Abstract. *The objective of this paper is to describe the implementation of a crowd animation module for Unity, utilizing Data-Oriented Technology Stack and a General-Purpose Graphics Processing Unit (GPGPU). The methodology adopted was based on evaluating the features from the traditional module and on an experimental process of porting them to an Entity Component System environment where critical performance algorithms were moved into the GPGPU. The presented module is able to animate several times faster than the traditional module while keeping the same visual outcome, although introducing some limitations to do so.*

Resumo. *O objetivo desse trabalho é descrever a implementação de um módulo alternativo para animação de multidões no Unity, utilizando Data-Oriented Technology Stack e uma General-Purpose Graphics Processing Unit (GPGPU). A metodologia adotada se baseia em avaliar recursos presente no módulo da Unity e em um processo de portar elas para um ambiente Entity Component System, em que partes críticas para performance foram movidas para GPGPU. O módulo desenvolvido é capaz de animar inúmeras vezes mais rápido que o módulo tradicional ao passo que mantém o mesmo resultado visual, embora introduza limitações para alcançar tais resultados.*

1. Introduction

In real-time rendering applications, the animation of crowds is an important visual effect that can be found in the game industry. Crowds can be used in a variety of applications, for example, to populate virtual scenes like forests, cities or villages. According to Nguyen (2007), the rendering process in video games has become more complex over time. Therefore, it's important that computing resources should be used efficiently.

In conformity with Lewis and Jacobson (2002), a game engine is a set of modules in charge of handling resources that are commonly related to games. They can reduce the time and cost of game development as new projects don't need to implement the basic features once the game engine, which can be used in multiple projects, has already done it.

However, sometimes the imposed design of a built-in game engine module isn't desired for certain projects, which leads to the development of an alternative module.

This article presents CrowdMorph, a module for Unity with the intention to optimize animation for simulated crowds of skeletal creatures by utilizing Data-Oriented Technology Stack (DOTS), a combination of technologies that work together to deliver a high performance, multi-threaded data-oriented approach to coding in Unity—and ComputeShaders, which are shaders capable of running General-Purpose Graphics Processing Unit (GPGPU) algorithms.

The module also converts relevant data from the built-in one, so users can keep using the editor tooling and a similar design for integrating animation into their projects. This conversion is required because, in order to employ technologies from DOTS, data must be in a data-oriented model as opposed to the traditional object-oriented model.

1.1. Main Objective

Implement a crowd animation module for Unity with Data-Oriented Technology Stack and a General-Purpose Graphics Processing Unit.

1.2. Specific Objectives

- Project the module components, following a data-oriented design and focusing on implementing as much as possible of the critical performance sections of the module in GPGPU shaders, like the mesh skinning and animation transformation processes.
- Implement the code responsible for converting the data from the traditional module into a data-oriented model, so they can be accessed by the animation phases.
- Use algorithms to achieve the same visual outcome from the traditional module, so module migration can be done with minimal impact on visual results.

2. Theoretical Background

This section introduces the main concepts required for the understanding of this paper. These are: Memory in modern computers, Data-Oriented Design, Entity Component System, Unity, Data-Oriented Technology Stack and Linear Blend Skinning.

2.1. Memory in modern computers

Consistent with Handy (1998), there's a variety of memory types. In that regard, each one has its own specifications where some are faster and others are cheaper. The faster the memory, the greater its cost. Modern computers combine multiple memory types into a memory hierarchy, so they can have a small amount of faster and expensive memory, while at the same time having a great amount of a cheaper and slower one. Consequently, this hierarchy can offer both speed and wide capacity.

In consonance with Handy (1998), the most commonly found volatile memory types in a modern computer are:

- Registers: they are the most expensive and fast memory. It's part of the processor itself and, mostly, it's used to hold operands, instructions and relevant addresses for the processor. They are directly accessed by the processor.
- Main memory: Usually made out of Dynamic Random-Access-Memory (DRAM), which is the cheapest form of semiconductor memory. This memory is much slower than the previous one, but it can offer a great amount of capacity due to its cost.

- Cache memory: It remains between the main memory and registers and is generally made out of Static Random-Access-Memory (SRAM), that is faster than DRAM, but much more expensive. Cache may be composed of multiple levels.

Fetching data in a modern computer, as stated in Smith (1982), consists of attempting to get the desired data from any of the cache units. If the data is not available, then it will have a cache miss and will move onward to the main memory and fetch a chunk of data instead of just the desired data. This process is called prefetching. It is done to store the chunk in the cache units, making it temporarily available for upcoming fetches. Once the chunk is stored in the cache memory, then it's referenced as a cache-line. Typically the cache-lines are 64-bytes wide.

The prefetching operation, as described by Vanderwiel and Kilja (2000), increases processor performance as it take less clock cycles to fetch the data from the main memory. Depending on the data memory layout, a cache-line can contain data that will not be used by the processor. This event is referenced as cache pollution and it hits performance because it will cause the main memory to be used more often.

2.2. Data-Oriented Design

In agreement with Fabian (2013), Data-Oriented Design (DOD) is a programming paradigm centered around data. Most of the programming paradigms pretend the hardware doesn't exist and shouldn't be taken into consideration in the application development. Despite of that, DOD motivates programmers during development to be aware about the hardware, for example, the cache memory is one of the most important concerns in the paradigm. This hardware awareness can contribute to performance as features intended to improve it aren't ignored.

The fundamentals of traditional Object-Oriented Programming (OOP), which is the most popular paradigm in the game industry, oppose this motivation. This is the cause of a series of pitfalls in the OOP that make the DOD a good alternative to it. The following items describe some of the most significant pitfalls, as reported by Fabian (2013):

- Object memory locations have no relation. Whenever an object is instantiated, the application will look for a free memory location to allocate it. This means that the data is likely to be spread in memory, without a logical order. Consequently, the cache miss rate is expected to increase.
- The containing classes of an object associate its data to a context, and operations within a context may require data that will only be used by certain operations. This a factor that increases cache pollution as the object contained all this data, even though most of the operation with the object may only require a portion of it.
- Related data most of the times will not stick together in memory. As a consequence, operations won't be able to take advantage of Single Instruction/Multiple Data (SIMD) instructions.

Data-Oriented Design solves these problems, mostly, by using different memory layouts where cache misses and cache pollution can be avoided greatly. SIMD instructions can be used to transform the data and the data transformation can be computed in parallel.

In modern computers, processors often have a set of instructions to process multiple data with a single instruction. It uses a series of parallel circuits to process multiple data [Flynn and Rudd 1996]. These instruction sets are called Single Instruction/Multiple Data (SIMD) instruction. They act in multiple data but their cost is almost the same as scalar instructions, which act on a single value at a time.

CrowdMorph is implemented using Entities, which is a package provided by Unity for developing games using Entity Component System, an architecture pattern that follows the Data-Oriented Design.

2.3. Entity Component System

Entity Component System (ECS) is an architecture pattern that follows the Data-Oriented Design. In OOP, data and logic are coupled within a object, but the ECS splits them into components for data and systems for logic. The entity is basically an identification or a key for a set of components. Systems are responsible for processing entities with certain components. For example, a VelocitySystem queries entities with Velocity and Position components every frame in a game to update their position. [Härkönen 2019].

Components are generally allocated in homogeneous contiguous memory blocks as Structure of Arrays. This allows systems to use the memory cache efficiently and split this block to be processed in parallel by worker threads, as well as to use SIMD instructions.

2.4. General Purpose Graphics Processing Unit

Initially, Graphic Processing Unit (GPU) was designed to move and color triangles. On top of that, operations like matrix multiplication and interpolation are faster in GPU compared to in CPU. Furthermore, as its purpose is to render massive amount of triangles, it's also designed to run instruction in parallel as much as possible using SIMD instruction and thread synchronization. General Purpose Graphics Processing Unit (GPGPU) allows the GPU to be used to general purpose data that aren't part of the traditional rendering pipeline [Metelitsa 2005]. CrowdMorph uses GPGPU to process animations as it consist of massive matrix operations.

2.5. Skinning

For performance, animations generally don't directly affect the meshes. Rather, a underlying skeleton is used to deform them. Instead of storing the animation data for every vertex in the mesh, skeletons only require animation data for few bones, which reduces the animation computing cost and memory usage, with each vertex in the mesh being influenced by one or more bones. Also, bone transformations are transferred to the their respective influenced vertices after they have been animated. This process of mesh deformation is called Skinning [Jacka et al. 2007]. Unity uses a skinning technique called Linear Blend Skinning, which is the same technique used by CrowdMorph.

2.6. Linear Blend Skinning

Conforming to Jacka et al. (2007), Linear Blend Skinning or Skeletal Subspace Deformation is the most popular technique used for skinning. In this algorithm, a skeleton consists of an array of matrices that represent the bone transformation matrices. The bone

transformations, when the skeleton was attached to the mesh, which are referenced to bindpose, are used by the algorithm to transform the vertex vectors from the model space to the bone space. For each bone that influences a vertex, a bone index and a weight are associated to the vertex. This algorithm may be implemented as the Figure 1 suggests.

```
float3x4 _BindposeInverseMatrices[64];
float3x4 _SkinMatrices[64]; // bone transformations

void lbs(int4 boneIndices, float4 boneWeights, float3 positionIn, out float3 positionOut) {
    for (int i = 0; i < 4; ++i) {
        int boneIdx = boneIndices[i];
        float3x4 skinMatrix = mul(_SkinMatrices[boneIdx], _BindposeInverseMatrices[boneIdx]);
        float3 vtransformed = mul(skinMatrix, float4(positionIn, 1));
        positionOut += vtransformed * boneWeights[i];
    }
}
```

Figure 1. Linear Blend Skinning Implementation (in HLSL)

2.7. Unity

Before introducing Unity, it's important to mention a conflict between two terms that will be widely used in this work. Although, the term "component" was used early to refer to the concept introduced by Entity Component System, in this section this term will strictly be referring to a new concept. In further sections, these terms will be distinguished between "authoring component" and "entity component", the first one referring to the concept that will be introduced, and the second to the already introduced concept.

This section will cover the Unity game engine. The research conducted here is mainly from the Unity Documentation[1]. Additional sources will be cited when necessary.

In compliance with Lewis e Jacobson (2002), a game engine is a collection of modules that handle resources that are commonly related to games, such as animating, rendering, networking, path finding, input handling, audio, physics, among others. Additionally, some game engines are accompanied by tools to improve the game development. These can reduce the time and cost of game development as new projects don't need to implement the basic features once the game engine has already done it. And, as mentioned earlier, some engines also provide tools, which contribute to productivity.

Unity is a cross-platform game engine that has a component-based architecture with Object-Oriented Design. Components, in this context, are classes which encapsulate logic and data for different concern, namely physics, animation, rendering and others. They must be attached to a GameObject which acts as containers for them. GameObjects represent characters, props, scenery and a number of different things. It can also be organized in hierarchies, in other words, each GameObject can have a parent. In addition to that, the last key concept are Assets, objects that can be serialized into files and later loaded into memory whenever they are requested. Assets can be created outside of Unity, like 3D models, animations or textures, and also within it, for example AnimatorControllers, Materials or even instances of an user custom type [Unity Technologies 2022].

[1]Unity Documentation [Unity Technologies 2022]

Another essential part of the engine is the Unity Editor. This is the Unity development environment. According to Unity Technologies (2022), it provides tools to create applications with the engine, including interfaces for setting up content, constructing levels, profiling, building and others. Additionally, Unity provides an Application Programming Interface (API) for extending and customizing the editor.

2.7.1. Unity's animation module

Since this article covers a module that is based on and compatible to Unity's animation module, this section will be used to describe features and concepts from their module for a better understanding of how they are related to the module proposed one in this work.

As described in the Documentations [Unity Technologies 2022], the animation module features the Animator, a component that is responsible for assigning animations to a game object hierarchy. This component can animate component data in the entire game object hierarchy. It can also call methods inside component instances. This feature is called animation events. They are associated to an animation frame and when it's assigned to the game object, the Animator component call every method in the components within the hierarchy where the method name matches the event name.

Animation clips store data typically using a set of curves that are utilized to describe how the properties in the hierarchy animate over time. An animation curve contains the path within the hierarchy to a property (e.g. LeftUpperArm/LeftLowerArm/LeftHand/m_LocalPosition.x), the component type that contains the property and the curve that can be evaluated to get the property value at a specific time.

Sometimes animation clips can be represented as dense clips. Instead of using keyframes to describe the curves, which are more expensive to the processor due to keyframe evaluation, dense clips store curves as arrays that contain the evaluated values for every frame in the animation. In a dense clip, the evaluation is faster as it only consist of accessing the two frames that the provided time is in-between and linearly interpolating their values [Unity Technologies 2022].

After the skeletal animation evaluates the bone rotations, scales and translations, it's necessary to compute the bone LocalToRoot matrices. This matrix can transform vertex vectors from bone space to the root space. It will be later used by the deformation phase (in the rendering module) for skinning.

Consistent with the Documentation [Unity Technologies 2022], Animator component requires a reference to an AnimatorController which holds the references to the animation clips that can be assign to the game object and defines how and when to blend and transition between them.

The AnimatorController is multi-layered to allow a set of state machines individually produce a transient animation for the object, so the controller can compose a more complex animation using these transient animations. The following items are some features present in the controller:

- AnimatorControllers contain parameters, representations of values that will be provided at run-time by the user to the Animator component.

- There are four types of parameters: ”Bool” (boolean data type), ”Float” (floating-point number), ”Integer” (integer number) and ”Trigger” (boolean data type but it has a special behavior: if a transition that requires a trigger to be true started, then referenced triggers are set to false).
- A state machine is used by layers to produce the transient animation. It consists of states and transitions where transitions allow a smooth blend between states in which states are associated to one or more animation clips.
- States can reference parameters that will be used to influence their behaviors. An example is the ”Speed Parameter”, that uses the referenced parameter value as a multiplier for the playback speed of the animation produced by the state.
- The state machines used by the layers have their transitions activated by a set of conditions within the transition itself.
- A transition condition is a structure that consists of three entries: the parameter name that should meet the condition, which operation (”If”, ”Greater”, ”Less”, etc.) should be used to validate it, and, sometimes, a value to be compared against the parameter value.
- Layer animations can be applied to the AnimatorController's composite animation with two different blending modes: ”Additive” and ”Overridden”.
- Additive blending mode allows animated properties of animations to be added between themselves. For example, the blending between a character animation where its head is rotated 10 degrees and another animation with 30 degrees, with both rotations to the same direction, would result in an animation with a head rotation of 40 degrees in that same direction.
- Overridden blending mode allows animated properties of animations to be overridden by properties from another animation. Following the previous example, the 30 degree animation would override or, in other terms, take the place of the 10 degree animation, making the final result be a 30 degree head rotation of the character.
- An Avatar Mask (a map from bone paths to bone influences) can be assigned to a layer, so it only influences a portion of the skeleton bones. For example, “Shoot” and a “Run” animation with a torso mask could be combined to allow the character to run while shooting.

2.7.2. ComputeShaders and ComputeBuffers

The ComputeShaders and ComputeBuffers API allows users to use GPGPU algorithms in the engine. This feature doesn't work on every platform that Unity supports. It requires certain graphics APIs, for example, DirectX 11 or above and Vulkan, as mentioned in the Documentation by Unity Technologies (2022).

The Documentation [Unity Technologies 2022] defines a compute buffer as an object that represents a buffer that remains in GPU memory. It is possible to set and get the buffer data, but getting the data is a very expensive operation, which mostly will result in a performance worst then keeping and processing the data in the CPU.

Compute shaders are the GPGPU algorithms itself. They can have their global values assigned in the CPU as well as the buffers used by it. Once a compute shader is dispatched, it will only be executed after all the previous dispatched shaders

that depend on any of the buffers that it's also using have already been executed. [Unity Technologies 2022].

2.8. Data-Oriented Technology Stack

Similarly as the previous section, the source of information about this section is the Unity Documentation, as well as the source code from the cited packages.

According to Unity Technologies (2022), Data-Oriented Technology Stack (DOTS) is a combination of technologies that work together to deliver a high performance, multi-threaded data-oriented approach to coding in Unity [Unity Technologies 2022]. These technologies are provided by packages that can be installed in the engine. The two most relevant packages used by CrowdMorph are the Burst Compiler and the Entities.

Burst Compiler translate Intermediate Language (IL)[2] bytecode to highly optimized native code. But the compiler has its restrictions: managed objects (objects handled by the garbage collector) are not supported. This means classes and arrays are not supported by the compiler as they are managed objects. Due to this restriction, assets used by the traditional module should be converted to a supported format [Unity Technologies 2022].

As mentioned in the Unity Documentation [Unity Technologies 2022], Entity Component System (ECS) is the core of the Unity's Data-Oriented Tech Stack. Entities is the package that provides a framework for developing applications using the ECS architecture.

Entities are basically a unique identification like a key in a relational database. They are associated to archetypes, which define the components they have. In other words, entities with distinct set of component will belong to different archetypes. Archetypes store the component data of its belonging entities in one or more archetype chunks. This container is a contiguous memory block where its layout is illustrated in the Figure 2.

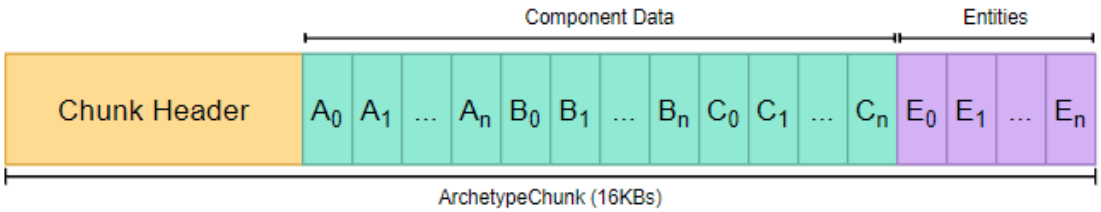

Figure 2. Overview of Archetype Chunk in Memory

As demonstrated in Figure 2, components are arranged as a Structure of Array (SoA). This results in more homogeneous and contiguous memory blocks in the application, which is the ideal for efficient cache usage [Handy 1998]. An archetype may have more than one chunk as they have a fixed size. This splitting of entities into multiple chunks is used to distribute entities to worker threads, contributing to make the application more suitable for parallel computation.

[2]"It is a product of compilation of code written in high-level .NET languages." [Microsoft 2021]

Component data generally remains in chunks, but there are two types of component that don't necessarily follow this rule. Firstly, the buffer component acts like a list, allowing the user a way to add multiple entries of the same component to an entity. Sometimes the buffer component capacity is reached. When this occurs, the entries in the buffer aren't stored in the chunk anymore, but in an external buffer exclusively allocated for it. Also, shared components can be used by multiple entities and are allocated in a separated buffer [Unity Technologies 2022].

In consonance with Unity Technologies (2022), System are objects (OOP) that update entities by transforming their component states, adding and removing components, in addition to managing the entity life cycle. System can also interact with other engine components, for example, a rendering system may request the GPU to draw a model in the scene every frame. Additionally, they can update entities in a multi-threaded environment.

3. Related Works

Nguyen (2007) developed an animation package using DirectX 10 (DX10). He stored animation data in texture pixels due to DX10 limitations, which aren't required anymore in DirectX 11, because the new version supports access to compute buffers in the vertex shaders. Nowadays, vertex shaders can read data that was previously processed in/by compute shaders. This new feature is used in CrowdMorph as the bone transformations are evaluated by compute shaders. Besides that, the author has a common goal to this work. His application concern was to have as many as possible animated characters on the screen. Also, he used the same skinning algorithm, Linear Blend Skinning.

Dong and Peng (2019) proposed a solution for large crowd rendering that used techniques like Level Of Detail (LOD), in which objects rendered far away from the camera will use a less detailed version to be rendered, and View Frustum Culling, a technique where the application decided what should be or not rendered by checking if the object overlaps the camera view area. Both techniques are present in Unity's rendering package for Entities, which is responsible for rendering the entities animated by CrowdMorph.

4. Methodology

During the development of this work, some of the packages from DOTS were experimental. In this phase, there is a lack of documentation for them. The main sources for the comprehension of the package feature applications were the documentation itself, the Unity forum[3] and their source code.

During module development, the most important design choice was to produce a module where animations are merely cosmetic effects. The animation processing occurs mostly in GPU restricting CPU to access skeleton transformations. This decision created some limitations, but was extremely beneficial for performance.

Due to the module's experimental nature, alongside with the fact that it's a adaptation of the traditional module and inherits most of its design choices from it, its development wasn't based on any formal development methodology. Instead of that, it was a contiguous experimental development process of porting existing features from the built-in module to an Entity Component System architecture from Entities package alongside GPGPU shaders.

[3]https://forum.unity.com/forums/data-oriented-technology-stack.147/

The module uses the Entities framework, that adheres the Entity Component System (ECS), an architectural pattern that follows the Data-Oriented Design (DOD). Also, GPGPU algorithms are used, for the critical performance sections of the application, via ComputeShaders.

The first step of the development process was declaring the system requirements. The requirements presented are:

- The package should use Unity's DOTS packages as its foundation.
- Although they must follow a Data-Oriented Design, Data structures and behaviors of Unity's Animator authoring component must be replicated into the package, so users can keep using the editor tooling for Animator and so that they can have the advantages offered by the package as the data from the traditional model could be converted.
- Animation transformations must be applied to skeletons in GPU. A component system should produce commands that will be uploaded to the GPU. Thus, a compute shader must process them.
- Skinning must be done in the vertex shader.
- Parameters, data used to drive the behavior of state machines inside Animator, should be provided through a user-defined entity component where its fields are bound to parameters that have the same name.
- It should expose animation events that were produced in the game loop, so another systems component can implement code to react to them.

In order to analyze the performance of the module, a scene was created with a crowd of animated characters, and another scene was created to replicate the same visual outcome produced by the first one, but using the built-in module. The second scene was used as a parameter to compare the module optimization. In both scenes were measured the average frame time using different numbers of characters. Thereafter, the data collected was used in a graph for better comparing.

Every benchmark was done using the same system and Unity version. In this case the specifications were:

- CPU: Intel® Core™ i7-10700 @ 2.90GHz
- RAM: 16GB DDR4, CL16 @ 3200MHz
- GPU: NVIDIA® GeForce® RTX 2060, 6GB, GDDR6, REV 2.0
- Motherboard: Asus ROG Strix B460-G
- OS: Microsoft Windows 10 (Version: 21H2)
- Unity Version: 2021.1.18f1

5. Project

The Figure 3 illustrates an overview of the module concerns, which is separated into two sub-modules. The part that is deployed with the application build runtime (like the game executable) is responsible for animating entities indeed. The second part is the Hybrid. It converts the traditional components and assets to a version compatible with Entity Component System where this process is done in the Editor.

Both modules were developed using Entity Component System package. Component system are used to encapsulate the module steps in the game loop and conversion

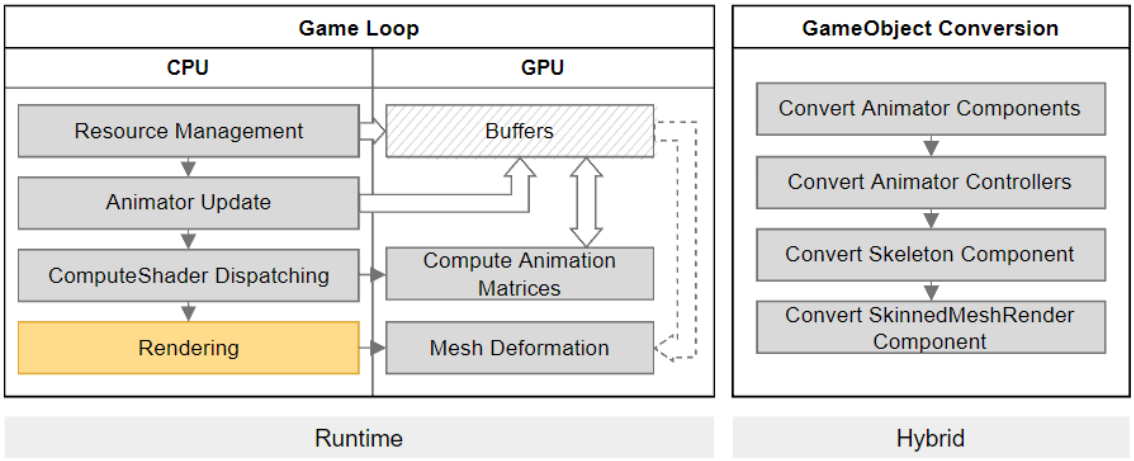

Figure 3. Module Concern Diagram

logic. The module is separated into two sub-modules: Runtime and Hybrid. The Runtime sub-module is actually the part of the module that is deployed in the game itself. Its concerns are contained in the game loop (every frame), as shown in Figure 3. The Hybrid sub-module is responsible for converting GameObject's authoring component to entity components, as well as the assets used by these components, such as AnimationClip, AnimatorController and AvatarMask.

5.1. Resource management

Initially, the concern is about Resource Management. Before an entity can be animated, the resources it will use must be allocated. This is done by a group of Systems. These Systems generally manage compute buffers (e.g. a buffer to store animation frames), resize the buffers to fit the content, upload data from CPU to GPU, reserve portions of the buffer to be used by entities and also provide buffer metadata, for example, where an animation clip have their frames allocated in the GPU.

The Resource Management Systems also stores compute buffer metadata related to a specific entity in components added to it, like the SkeletonMatrixBufferIndex entity component stores the first index within the skeleton matrix compute buffer where the block reserved to the entity remains. Additionally, the Systems are responsible to free space in the buffers as long as it's not being used anymore, for example, when an entity is destroyed and its skeleton matrices will no be used anymore.

5.2. Animator update

After Resource Management, Animator Update is the phase where entities' Animator state is updated. A single system is responsible for this concern, the UpdateAnimatorSystem. It has the function of processing entities in order to reproduce the Animator authoring component behaviors, replicating most features cited in the Section 2.7.1. During this process, the System stores the Animation Events produced in a buffer, so users can implement Systems to react to them. The second buffer is the Animation Commands buffer. It produces commands for the animation compute shader, which will process them.

For compute shaders, it's important that threads are synchronized for performance [Metelitsa 2005]. This means that every thread should be executing the same instruction

at the same time. However, a single entity can produce multiple commands per frame, for example, it can have multiple layers that are creating commands. Within these layers, there may be transitions in progress, which would result in two commands: one for the current state animation and another from the next state. If all the commands produced in a frame were stored in a single array and sent to the computer shader, it would cause unpredictable behaviors because more than one thread could be trying to write/read to the same skeleton concurrently.

To avoid this, the commands are split into batches, that should only contain at most one command per entity. This allows the compute shader responsible for processing the commands to be dispatched multiple times and, as they have common dependencies, Unity will execute a dispatched compute shader only after the previous one has been completed [Unity Technologies 2022]. With this approach, it's possible to control command execution order to a skeleton.

Another obstacle was skeletons with different bone count. As the animation computation iterate the bones to apply the animation transformation, it would result in idle threads due to thread synchronization [Metelitsa 2005]. The solution to this was grouping commands by bone count. Thenceforth, the groups must be dispatched separately, so that each dispatch only do the processing of commands that share the same bone count.

5.3. Compute shader dispatching & Compute Animation Matrices

In this phase, compute shaders are dispatched. There are three compute shaders that will be dispatched:

- ComputeAnimation – This processes the commands, applying the animation to the skeletons. Multiple instances of this shader will be dispatched, as discussed earlier, to allow multiple commands to the same skeleton per frame.
- ComputeLocalToRoot – When animation are applied to the bones in skeletons, they are in their local space, which is the space relative to the bone origin itself. Yet, for deforming the meshes, they should be in local to root (the multiplication of its matrix by each of its parent matrix). This process is done by this shader.
- ComputeSkinMatrices – When a mesh uses the skeleton matrices, it don't actually accesses them directly. This occurs in a secondary buffer called Skin Matrices. That's why the meshes reference bones in their particular order. This shader will copy the referenced skeleton bone matrices for each skinned mesh instance, allowing multiple skinned meshes per skeleton.

5.4. Rendering & Mesh Deformation

The Rendering concern has a different color in Figure 3 than the others. That's why it is handled by the Unity's rendering package for Entities, Hybrid Renderer. CrowdMorph provides a node for Shader Graph—Unity's package for shader visual scripting [Unity Technologies 2022]—that allows mesh deformation using the skin matrices previously computed. The node uses Linear Blend Skinning for the skinning.

During this phase Unity's rendering component systems upload data from entities components to the GPU so they can be accessed in vertex/pixel shaders. For example, an entity with rendering component can upload user-defined components to GPU, just

assigning a C attribute to its declaration. This feature is used by CrowdMorph to access bone transformations (previously processed by compute shaders) during the vertex shaders, which are used to deform the mesh.

5.5. Hybrid

Entities are encouraged to be converted from GameObject, so data can be stored in a more human-like model and be converted to a closer machine-friendly version. The Entities package provides an extensive API for this conversion. CrowdMorph decomposed authoring components into multiple entity components, for example, the authoring Animator component is decomposed into AnimatorLayerState, SharedAnimatorController and a user-defined entity component to store the Animator parameters.

This approach to converting GameObjects permits users to keep setting up animated 3D objects as they normally do, with a few extra steps to be animated by CrowdMorph.

5.6. Unsupported Features

CrowdMorph partially port the features present in Unity's module. This result in limitations, for example it doesn't support every feature present in the traditional Animator authoring component. In reason of the module's processing of the animations on the GPU, this data isn't accessible on the CPU, so component systems can't get bone transformations of a skeleton, which is sometimes required, for example, to place a weapon in the hand of an animated character.

The decision to keep the animation data in the GPU also increases the complexity to add new features to it, like Inverse Kinematic (IK), which is a very common technique used along animations. Furthermore, features like Humanoid animations, BlendTree motions and state machine behaviors are not supported.

5.7. Results

The purposed objectives in this work were achieved. The module was developed using Entity Component System package and adhered the use of GPGPU for critical performance section, like mesh deformation and skeleton transformations. It converts GameObject and assets from the traditional model into one module that supports ECS. Finally, animations produce the same visual result in CrowdMorph and Unity's module.

To compare the module to Unity's one, two levels were created in the engine, where they produce the exactly same visual result; but, each level is using a module. The level consists of a group of animated characters, with the average frame per second in the level being measured several times with different number of characters. Then, these measures were used to create the graph in the Figure 4.

The Figure 7 shows the characters used in the scenes. The Entity (left) was completely converted from the GameObject (Right). With some extra settings, the GameObject was able to be converted. The Entity produced by this conversion achieved the same visual result as the GameObject. Both characters used in the scenes switch between multiple animations. For a better visualization a video was upload to YouTube[4].

[4]https://www.youtube.com/watch?v=ZUkGL5S8-RQ

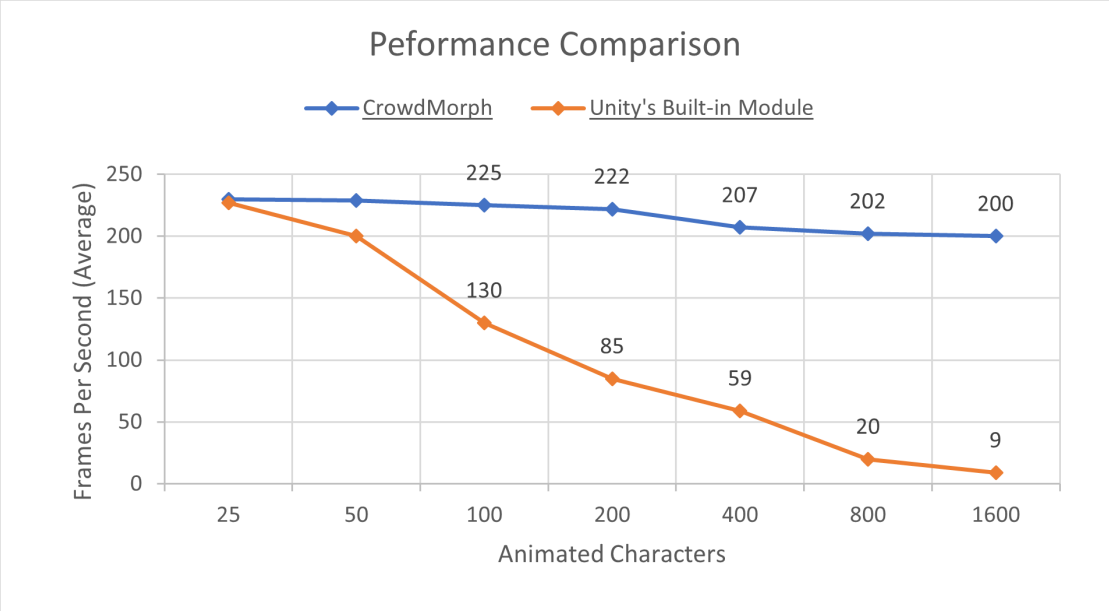

Figure 4. Performance Comparison Graph

Figure 5. Entity and GameObject

The system used to test the module was able to animate 1,600 characters with an average of 200 frames per second (FPS), while Unity's module produced an unplayable result of 9 FPS. Even at 400 characters, Unity's module was consuming too much time to allow any other game mechanic to work along the characters and keep the game running above 60 FPS.

The Figure 6 shows the compute shader implementation responsible for animating the skeletons, in other words, to apply the animation clip transformation to skeletons. It reads a command from the buffer and executes its. During the process, the shader will iterate the skeleton bones and respectively applying the clip bone transformations. The command assigns the type of blending should be used to process it. But the shader

```
[numthreads(64, 1, 1)]
void ComputeAnimationKernel(uint3 id : SV_DispatchThreadID) {
    const AnimationCommand cmd = _AnimationCommands[id.x + g_BatchCommandOffset];

    for (uint i = 0; i < g_BoneCount; i++) {
        const AffineTransform interpolatedKeyframeMatrix = GetClipInterpolatedKeyframe(cmd.ClipSampleBufferIndex, cmd.PackedKeyframes, cmd.KeyframeWeight, i);

        const AffineTransform previousSkeletonMatrix = _SkeletonMatrices[cmd.SkeletonMatrixBufferIndex + i];
        const AffineTransform additiveReferencePoseMatrix = _ClipSamples[cmd.AdditiveReferencePoseMatrixBufferIndex + i];

        const AffineTransform additiveMatrix = AffineTransformMul(AffineTransformMul(interpolatedKeyframeMatrix, additiveReferencePoseMatrix), previousSkeletonMatrix);
        const AffineTransform overrideMatrix = interpolatedKeyframeMatrix;
        const AffineTransform resultMatrix = AffineTransformLerp(additiveMatrix, overrideMatrix, cmd.BlendingMode == kBlendingMode_Override);

        const float weight = GetBoneWeight(cmd.SkeletonMaskBufferIndex, i) * cmd.Weight;
        _SkeletonMatrices[cmd.SkeletonMatrixBufferIndex + i] = AffineTransformLerp(previousSkeletonMatrix, resultMatrix, weight);
    }
}
```

Figure 6. Compute shader kernel responsible for applying animation to skeletons

processes both blending modes: additive and override. Because threads should be synchronized, therefore, all threads should execute the same instructions at the same time, so instead of using control flow instructions the blending mode result is selected using arithmetic instructions.

The skinning processes and the skeleton animation were moved to the GPU. The CPU is only responsible for processing the state machines producing animation commands to the GPU and animation events to be available to subsequent component system.

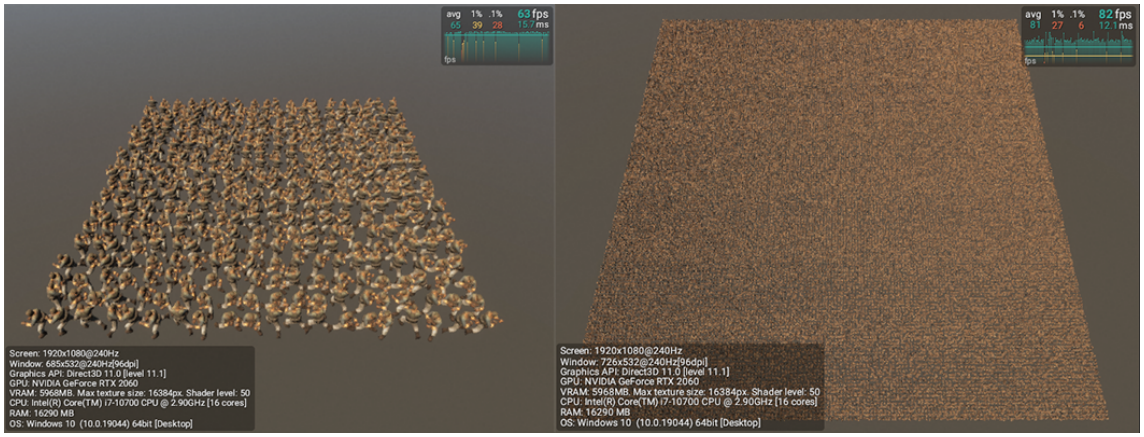

Figure 7. GameObjects (Left) and Entities (Right)

The Figure 7 shows two scenes. The left are 400 GameObject characters, while the right are 25,600 entity characters animated by CrowdMorph. Even though the right scene has 64 times more characters, it can achieve a higher FPS than the left scene.

Finally, the module is delivered as a package, so users can install it directly from a git repository. The source code can be accessed in the GitHub repository[5].

6. Conclusion and Future Work

CrowdMorph has been successfully developed. The process of analyzing the Unity module and porting its features was enough to achieve the desired results. CrowdMorph is capable of handling thousands of animated entities while keeping a reasonable computing cost. It opens the possibility for game designs to explore concepts that depend on animated crowds, like Real-Time Strategy (RTS) games.

[5]https://github.com/felipemcoliveira/com.felipemcoliveira.crowdmorph

The research encountered a some limitations. Like the lack of documentation for how the Unity's operates. Nonetheless, the limitation was overcame, as during the research, Unity released their animation module for Entities. Even though the package didn't have the same purpose as CrowdMorph, which is crowd animation, and their module didn't have any documentation, its source code was analyzed for a better understanding of Unity's internal concepts revolving animation. This contributed immensely to CrowdMorph implementation.

Another limitation was testing the module in development. In virtue of module results being motions that should be analyzed frame by frame, alongside with the fact that Unity didn't provide any built-in solution for imaged-based testing, the implementation could only be manually tested in the final stage of development, where enough module components were already working to produce a visible result.

The module also has features that weren't ported from Unity's module. These features include Humanoid animations, BlendTrees, state machine behaviors, among others. Nevertheless, it was designed to be able to receive some of these features in the future. Ultimately, optimizations aren't part of the next module updates, as its current state has achieved the desired performance.

References

Dong, Y. and Peng, C. (2019). Real-time large crowd rendering with efficient character and instance management on gpu. *International Journal of Computer Games Technology*, 2019.

Flynn, M. J. and Rudd, K. W. (1996). Parallel architectures. *ACM computing surveys (CSUR)*, 28(1):67–70.

Handy, J. (1998). *The cache memory book*. Morgan Kaufmann.

Härkönen, T. (2019). Advantages and implementation of entity-component-systems.

Jacka, D., Reid, A., Merry, B., and Gain, J. (2007). A comparison of linear skinning techniques for character animation. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 177–186.

Metelitsa, B. (2005). Comparing software development approaches for general purpose gpu computing. In *21st Computer Science Seminar*. Citeseer.

Microsoft (2021). What is "managed code"?. Available in: https://docs.microsoft.com/en-us/dotnet/standard/managed-code, Accessed in May 4, 2022.

Nguyen, H. (2007). *Gpu Gems 3*. Addison-Wesley Professional, first edition.

Smith, A. J. (1982). Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530.

Unity Technologies (2022). Unity documentation. Available in:https://docs.unity3d.com/, Accessed in May 2, 2022.

Vanderwiel, S. P. and Lilja, D. J. (2000). Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199.