CEIG - Spanish Computer Graphics Conference (2014)
Pere-Pau Vázquez and Adolfo Muñoz (Editors)

CAVAST
The Crowd Animation, Visualization, and Simulation Testbed

A. Beacco[1] & N. Pelechano[1]

[1]Universitat Politècnica de Catalunya

Abstract

Simulation, animation and rendering of crowds has become an important part of real-time applications such as videogames. Virtual environments achieve higher realism when being populated by virtual crowds as opposed to appearing uninhabited. There has been a large amount of research on simulation, animation and rendering of crowds, but in most cases they seem to be treated separately as if the limitations in one area did not affect the others. At the end of the day the goal is to populate environments with as many characters as possible in real time, and it is of little use if one can for instance render thousands of characters in real time, but you cannot move more than a hundred due to a simulation bottleneck. The goal of our work is to provide a framework that lets the researcher focus on each of these topics at a time (simulation, animation, or rendering) and be able to explore and push the boundaries on one topic without being strongly limited by the other related issues. This paper presents therefore a new prototyping testbed for crowds that lets the researcher focus on one of these areas of research at a time without loosing sight of the others. We offer default representations, animation and simulation controllers for real time crowd simulation, that can easily be replaced or extended. Fully configurable level-of-detail for both rendering and simulation is also available.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

1. Introduction

Crowds simulation [PAB08, TM13] is becoming more and more important in computer applications such as building evacuation planning, training, videogames, etc., with hundreds or thousands of agents navigating in virtual environments. Some of these applications, in order to offer complete interaction with the user, need to be run in real-time. In order to achieve natural looking crowds, the simulation needs to exhibits natural behaviors, to have characters that are animated with smooth and realistic looking walking styles, and also that the rendering does not exhibit any noticeable artifacts. When simulating large numbers of characters the goal is usually to produce both the best local motion and animation, minimizing the awkwardness of movements and eliminating or lessening any visual artifacts.

Achieving simulation, animation and rendering of the crowd in real-time becomes thus a major challenge. Although each of these areas has been studied separately and improvements have been presented in the literature, the integration of these three areas in one real-time system is not straight forward. There are some commercial tools that provide aids to simulate crowds, but in most cases there are limitations that cannot be overcome, such as finding bottlenecks in a different area than the one you are researching on, thus pushing you from meeting real time constraints as we increase the size of the crowd.

Due to the high computational needs of each of these three areas individually, the process of integrating animation, simulation and rendering of real time crowds, often presents trade-offs between accuracy and quality of results. As we will describe in this paper, these three areas cannot be treated in a completely separated way as most current tools do, since there is a strong overlapping between them, and users need to be aware of this when setting up a simulation (see Figure 1). For example, we cannot increase the number of animations easily if we are rendering exclusively with impostors.

Currently it is not easy to find a real time framework that allows you to easily work on one of these areas. If for example you want to focus your research on a new steering behavior, you will start and do most of your experiments by visual-

DOI: 10.2312/ceig.20141108

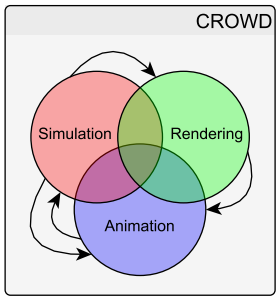

Figure 1: *Simulation, Animation and Rendering of Crowds are three overlapping research areas dependent on each other which are continuously interacting.*

izing a set of circles or cylinders representing your different agents. But in the end you would like to see your results represented by 3D animated characters. The switch from having 2D circles to fully articulated 3D characters in real time can be very time consuming. Or for example, your research interest may be focused on implementing a new representation for rendering thousands of characters. Once you achieve real time visualization of such a huge amount of characters, you do not want to end up displaying them in a grid formation or giving the agents random positions where they stay in place. Instead, one would want them to be animated and moving in a virtual environment, if possible with collision avoidance and natural animation. Again this is not straight forward in current frameworks. And finally, the same applies for the animation field, if you are defining new animation controllers for 3D characters, you would like to test them with hundreds of characters moving around a virtual environment with realistic rendering.

In this paper, we present a novel framework that embeds these three elements: Simulation, Animation and Rendering of crowds. Each of them presented in an independent-but-linked modular way. The final tool becomes then a prototyping testbed for crowds that allows the researcher to focus on one of these parts at a time without loosing sight of the other two. This tool could also be very handy for introducing crowd simulation in the classroom. Our bundle includes some basic resources such as character models, libraries and implemented controllers interfaces, which allows the researcher to have a basic crowd simulation engine to get started, and to be able to focus exclusively in a particular area or research. Our module also lets the researcher to have communication and interaction between these areas, if he desires to treat some of them in a more dependent fashion. We offer default representations, animation and simulation interfaces and controllers in a modular way, that can easily be extended with your new research work. We also include a fully configurable system of level-of-detail for animation, visualization and simulation.

2. Related Work

There are some well known commercial tools for modeling and simulation software such as 3D Studio Max [Auta], Maya [Autc] or Blender [BF] that allow us to add crowds to a virtual scene. There exist also many plug-ins for these packages that can be used to extend their basic features. For example Golaem Crowd [Goa14] is a powerful plug-in for Maya [Autc]. Golaem Crowd is a complete commercial package for crowd authoring, including tools for placing crowds, create behaviors, animate characters, create diversity of agents, and render the resulting simulations. Although they offer some real-time previsualizations in order to help the artists creating new crowds and defining behaviors, their target is mostly the movie industry producing high quality offline renders. It is not as much a research platform as it is a commercial production tool. Also in the movie industry, a major competitor is Massive [Mas14], an expensive crowd simulation tool that has been used in many films.

In the crowd simulation literature there are some steering behavior libraries such as OpenSteer [Rey], which also includes a demo software for simple visualizations of the implemented steering behaviors through simple 2D representations. We can also find SteerSuite [SKFR09], a flexible but easy-to-use set of tools, libraries, and test cases for steering behaviors. The spirit of SteerSuite [SKFR09] was to make it practical and easy-to-use, yet flexible and forward-looking, to challenge researchers and developers to advance the state of the art in steering. Although their simulation part is very complete, there is not such thing as animated 3D characters for visualization of the simulated crowds.

For character animation, there is just a few libraries that we can easily include in any C++ project. A commercial example would be Granny 3D [RGT], which includes a complete animation system including features such as blend-graphs, character behavior, events synchronization or procedural IK. But again it is a solution for artist and for commercial products, not for doing research. Some free solutions are Animadead [But], a skeletal animation library with basic functionalities, or Cal3D [Cal] another skeletal animation library which also includes an animation mixer and integrates morph targets. The Hardware Accelerated Library for Character Animation, HALCA [Spa], extends the Cal3D library to include new features such as GLSL shaders support, morph animations, hardware accelerated morph targets (blend shapes), dual quaternion skin shaders, JPEG texture files, direct joint manipulation and other additions. The FBX SDK [Autb] is another library allowing to read FBX files, which is a widely used and extended format for character modeling and animation. The API includes some skinning examples, but you should still program your animation library to use FBX models imported with it. Our framework currently offers HALCA [Spa] as the animation library, but the user can create its own characters based on any other library.

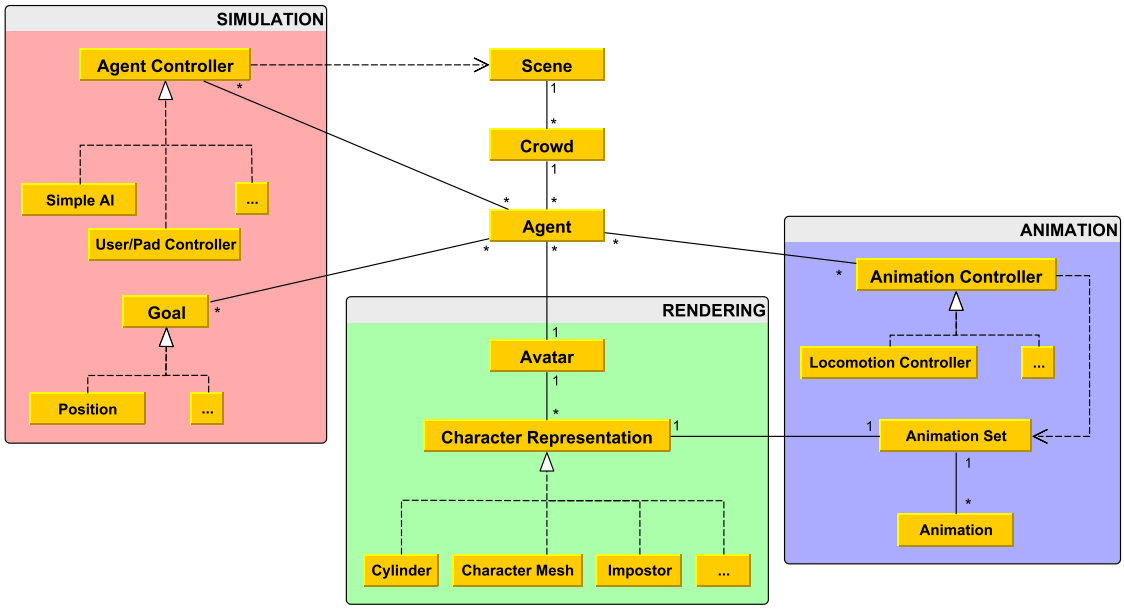

Figure 2: *Diagram with a rough overview of the classes in CAVAST.*

In the case of applications that require real time crowd simulation, such as video games, there are several tools commonly used both commercial (Unreal [Epi14], Unity [Uni14] and GameBryo [GU]) and open source (Ogre [Ogr14] and Panda 3D [CMU]). Unreal [Epi14] is a widely licensed game engine in the professional video game industry, with powerful and refined tools. Unity [Uni14] is a newer game engine that is also used for professional games, although it is more widely extended in the indy game development community. It offers a render engine, a complete animation system called Mecanim, with a very user-friendly interface for authoring state machines (such as blend trees) and retargeting capabilities. Mecanim also includes modules for steering behavior and navmesh generation. It is relatively easy to start a project and learn how to work with it, and a lot of researchers are starting to use it. But it still remains a commercial game engine, and you have to develop your extensions using scripts. You can use your own C++ code, which is faster, but you need to implement plug-ins and wrappers for them. Gamebryo [GU] is a similar product, modular and extensable, but still focused on game design. Ogre [Ogr14] and Panda 3D [CMU] are open source graphic render engines that include some features like animation systems or simple AI modules, which can be easily extended.

With most of the systems described above, you will find severe limitations when trying to scale up your work. For instance you may have developed a new rendering technique for thousands of deformable characters, but the selected engine may only animate a few hundred in real time. Using Unity [Uni14] you might be restricted to a fixed renderer and to its animation system, unless you implement your own using plug-ins. Implementing and integrating rendering plug-ins to Unity is possible although not straight forward, and might require a pro-license. The rendering pipeline in Unity is not always clear, and our experience says you can not completely control the OpenGL state. Ogre [Ogr14] has more potential, and has a basic animation system, which should be improved in newer versions, but you still need to integrate it with your AI libraries. The learning curve of the Unreal Engine [Epi14] is hard, and might not be worth if you are aiming for research and not for a professional and commercial appealing result.

None of these solutions offers a flexible yet customizable framework for the research community to work with when it comes to real time requirements, giving them freedom to modify either simulation, animation, rendering or any combination of these parts. In addition to that you might not be willing to pay expensive licenses if you are only targeting research applications. There is though some research platform in the crowd simulation field that are worth to mention.

CAROSA [All10] is an architecture to author the behavior of agents, and obtain heterogeneous populations inhabiting a virtual environment. Its framework enables the specification and control of actions, and is able to link human characteristics and high level behaviors to animated graphical depictions. Although it does not include research tools for rendering, it is prepared to be used on an external software. ADAPT [SMKB13] is an open-source Unity library

delivering a platform for designing and authoring functional, purposeful human characters in a rich virtual environment. Its framework incorporates character animation, navigation, and behavior with modular interchangeable and extensible components. But since it is a library for Unity, it does not allow you to control rendering. Project Metropolis [OE11] aims to create the sight and sounds of a convincing crowd of humans and traffic in a complex cityscape. They also focus on exploring the perception of virtual humans and crowds, through psychophysical experiments with human participants. But Metropolis is a large and complicated research project, with tens of research goals, rather than a tool for researchers to get started working in the crowd simulation field. Similar to what we aim to do, SmartBody [Sha] is a character animation library that provides synchronized locomotion, steering, object manipulation, lip syncing, gazing and nonverbal behavior in real-time. It uses Behavior Markup Language (BML) to transform behavior descriptions into real-time animations. SmartBody is a good tool to develop and explore virtual human research and technologies, but it is focused on one character, or a small number of characters, and not on crowds as it is our desire. Thus our goal is to provide the graphics community with a tool in order to be able to quickly get started on a new research project related to crowd animation, visualization or simulation.

3. Overview

We present CAVAST: the Crowd Animation, Visualization and Simulation Testbed, a new prototyping and development framework, made for and by researchers of the graphics community specialized in crowds. The goal of this work is to provide a framework with state-of-the-art libraries and simple interfaces to ease the work of starting a project regarding simulation, animation or rendering of crowd. This framework provides a basis to start working in this field, focusing on solving a specific problem in simulation, animation or rendering without loosing sight of the other aspects, that is conserving the communication and interaction possibilities between them.

Figure 2 shows a rough overview of the classes and interfaces present in CAVAST. The scene render engine is going to need some basic information to render an agent. This includes at least: visual representation (even if it is just a 2D point), position and orientation.

The Agent class is the core class of our framework and it is linked to:

- One or more Agent Controllers. These controllers deal with the kind of simulation methodology used for pathfinding and local motion. It also needs an interface to assign and describe one or more Goals to the agents (into a queue).
- One Avatar containing the Character Representation used by the Rendering module. Notice that one avatar can be shared by many agents.
- One or more Animation Controllers, an interface class in charge of the Animation module to deal with skeletal animation.

Each one of these interfaces are described in more detail in the following subsections.

3.1. Simulation

The Simulation module needs to include at least an implementation for the Agent Controller interface. This module will be responsible for moving the Agent in the virtual environment. The Agent Controller consists of either an implementation of a behavioral model based on for example steering, social forces, rules, or any other model that includes the AI of the agent. Alternatively, it could be directly controlled by the user input through a User Controller. When the Agent is controlled autonomously, it is usually required to have certain goals. The type of goals required vary from one system to another. Our framework provides a Goal interface and a basic implementation consisting of just a Position. The user can expand the kind of goals by implementing new Goals such as a position with orientation, or a position with orientation and time stamp. The Agent Controller has access to the Scene in order to query information about fixed obstacles, dynamic obstacles and other members of the Crowd. This information can be used for collision avoidance for example. Notice that the Agent Controller can also integrate physics libraries to accelerate collision detection if needed.

3.1.1. Pathfinding

As for pathfinding, we provide a Pathfinder interface class and a basic implementation of the A* algorithm [DP85]. This could be easily expanded to new Pathfinder classes such as D* Lite [KL02], ARA* [LGT03], etc. This class works over a Graph interface class, which can be either a Grid or a Navigation Mesh representing the scene. A Pathfinder also might need a Heuristic to work with, in our case we have a simple Euclidean distance, but it could easily be any other function estimating a cost to reach a node of the Graph. Our current version only provides a Grid representation from a randomized generated scene (filling cells in a grid with obstacles), although we plan to include a navigation mesh creation module from any static 3D geometry loaded in the scene. So, if the Agent Controller has a Pathfinder, it will be used to find a path and generate intermediate goals (way points) to insert into the goals queue of the Agent. Figure 3 shows a diagram of the simulation module in more detail, but for the sake of clarity not all the classes have been included in it.

3.1.2. Agent Controller

The main method an Agent Controller has to implement is `exeSimulation(ref Agent a, float elapsedTime)`. This method will be in charge of actually moving the agent, and will modify attributes of

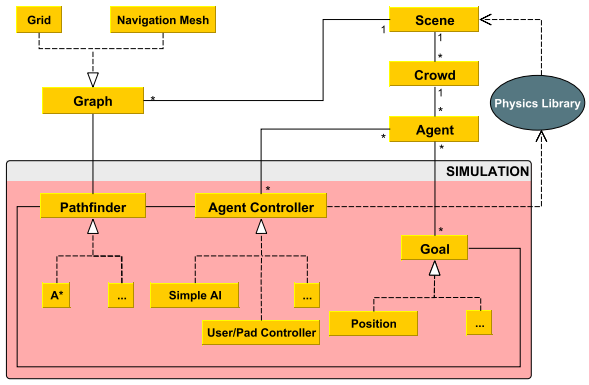

Figure 3: *The simulation module*

the Agent instance `a`, such as position, orientation, and velocity, as the result of executing forward the simulation during an elapsed time equal to `elapsedTime`. The other method that an Agent Controller needs is `getPathFinder()`, returning the PathFinder of the controller if there is any. A Pathfinder requires to implement a function `findpath(ref Graph graph, ref Node start, ref Node end, ref Heuristic h)`, returning a sequence of Nodes from `graph` representing a path from `start` to `end`. A Graph will be composed of Nodes, which can either be Cells in the case of a Grid or Polygons in the case of a Navigation Mesh. A Goal must have a function `isReachedByAgent(ref Agent a)` returning true when it is reached by an Agent instance `a`.

3.1.3. Crowd simulation

Transparent to the user, the Crowd will be in charge of iterating over all its agents. Currently, when an agent does not have a goal it will get one randomly assigned, although an Agent has a function to set up its current Goal. If the Agent Controller has a PathFinder, it uses it to find waypoints and insert them into the queue as intermediate goals. Once a goal has been reached, the next goal of the queue is the new goal to be used by the `exeSimulation` method. Then the Agent Controller executes its simulation for the elapsed time.

Figure 4 shows an example crowd of 200 cylinders moving in a random generated grid, using A* for path planning and Reynolds [Rey99] for steering, using CAVAST.

3.2. Rendering

An Agent has to be associated with an Avatar. We call an Avatar a collection of one or more Character Representations, along with the main dimensions we want for it, that is the size we want for our representations for each 3D axis. Having different representations allows us to use them for different levels of detail (LODs), and the main size of the Avatar allows us to be consistent between different representations. If for example we want to replace our character by some 3D model of a cylinder for far away agents, we will be able to scale the cylinder to the same dimensions as the original 3D mesh by scaling its bounding box. An implementation of the Character Representation interface must provide a Shader Program (with at least a Vertex and a Fragment shader) and the methods `render()`, to render it individually, and `instancedRender(int n, ref VertexBufferObject instancesDataBuffer)` in order to use instancing [Dud07b], efficiently rendering `n` instances using the corresponding data in `instancesDataBuffer` for each one of them. We also request for methods to get the bounding box, the bounding sphere and the bounding cylinder radius of the representation in order to help for collision detection and selection algorithms.

3.2.1. Scene Render

CAVAST uses its own scene library to manage the crowd scene. Its main modules are a Scene Graph represented by a Scene Tree, and a Transform class which is the nodes class of our Scene Tree. A Transform has a name and contains one absolute transformation matrix and one relative to its parent. A Transform can contain a Render Object, although it is not required and therefore a Transform can be empty (to perform relative transformations). If so, it also needs a Shader Program name to bind it before rendering. A Render Object is an interface class for the scene library to know how to render things in the Scene Graph, and one Avatar is a subclass of a Render Object. All of this is transparent to the user who wants to implement its own Character Representations. CAVAST and the Avatar class will be in charge of creating the proper transforms and add them to the scene.

When a Crowd *i* is added to the Scene, a Transform named "Crowd i" is added to the scene root. When adding agents with the same Avatar, their corresponding Transforms are grouped in a group Transform with the name of the Avatar. Inside that transform, agents are also grouped by Character Representations in a group transform for each one. When using level of detail, agents can change between Character

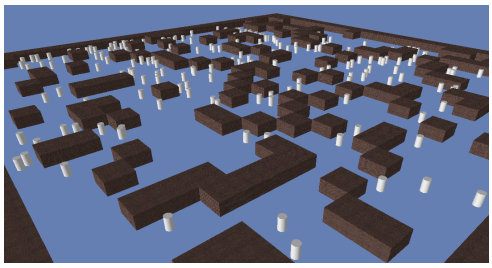

Figure 4: *An example simulation of 200 agents using CAVAST.*

Representations, and thus need to change between groups. This is dynamically and automatically carried out by the Crowd class. The main reason for doing this is to be able to perform instancing, and to accelerate the rendering of all the instances sharing the same Character Representation. Having all the Transforms for all the instances of one representation in the same group makes it fast to fill the instances data vertex buffer object with their individual data. Figure 5 shows an example view of a possible scene hierarchy.

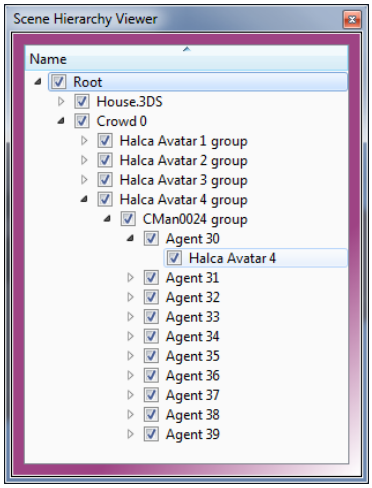

Figure 5: *An example view of a scene hierarchy.*

3.2.2. Character Representations

Our current system provides two Character Representations: a basic one based on just a cylinder 3D mesh (which could be easily extended by any 3D static mesh) and an animated Character Mesh in the Cal3D format [Cal] through HALCA [Spa]. The user could introduce any other Character Representations such as those based on impostors. As an immediate future work we plan to incorporate an automatic impostor generation module and their corresponding implementations of the Character Representation interface.

The constructor function should load the necessary geometry and resources (textures). A Shader Program should be loaded using our Shader Manager, and its name should be retrieved by the `getShaderName()` function (one of the Character Representation interface functions). The `render()` function only needs to send all the necessary information to the shader (via uniforms, attributes, or whatever you want to use) and to render the geometry. In addition, the `instancedRender(int n, ref VertexBufferObject instancesDataBuffer)` function needs to bind the vertex buffer object and enable any vertex attribute pointers necessary to render the different instances.

Notice that the shader binding is done by the scene library when rendering the corresponding transform of the agent. This allows the user to dinamically change the shader through the interface (whenever the new shader is able to handle the same data).

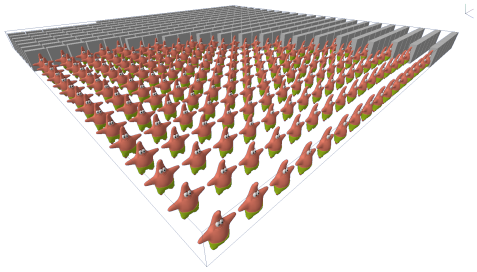

Figure 6: *Level of detail: 500 Avatars with a 3D static mesh and with a cube for agents at 30 meters or more from the camera.*

Figure 6 shows an example scene where 500 Avatars of the same type are represented using two Character Representations, a static 3D geometry for closer agents, and cubes for farther aways agents. And figure 7 shows 1000 agents represented using 4 different Avatars with only one Character Mesh for each one.

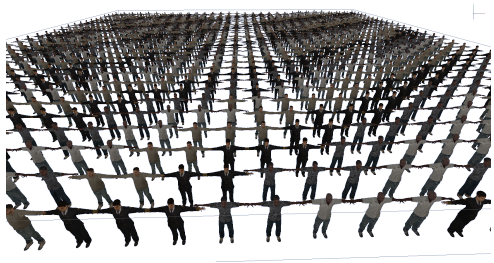

Figure 7: *1000 characters of 4 different types of Avatars using only one Character Mesh of around 5000 triangles for each one (without level of detail, nor animation).*

3.3. Animation

A Character Representation may be animated, that is, have a method of adapting the character representation to different poses. The most extended approach for animating 3D characters is skeletal animation [MTLTM88]. An alternative approach for mesh animation consists on having multiple deformed meshes (one per keyframe) which are switched rapidly to create the illusion of animation. You could avoid computing the deformation of a character mesh by storing pre-computed deformed meshes for each keyframe of the animation, and then carefully sorting these meshes to take cache coherency into account [UCT04]. Another further alternative to skeletal animation is morph target animation, where vertex positions are stored not only for the reference

pose, but also for each frame, or for each keyframe [Lor07]. Although impostors are easier for static objects, some works use impostors to render crowds of animated agents. Depending on how they are implemented, impostors animation can switch between different textures [TLC02], or may still use the skeletal animation [BAPS12].

3.3.1. Preprocessing Animations

Independently of the Character Representation chosen, if we want to animate it then we need to have an Animation Set. An Animation Set is an interface class offered by our framework, which is composed of different Animations (also called animation clips). The Agent can have an Animation Controller, in charge of selecting or synthesizing the best animation, in order to properly follow its current motion. The Animation Controller is thus dependent on the current Character Representation and on its Animation Set.

When using skeletal animation through an animation library, the animation controller will probably make an extended use of it. For example, we could have an Avatar linked to an Agent, whose main Character Representation is a character, rendered and animated with an accelerated animation library, such as HALCA [Spa]. That Agent could have a Locomotion Controller implementing an Animation Controller, which will be using that library to preprocess the Animation Set, by analyzing and extracting information from each Animation. Then at execution time, the Locomotion Controller could read the velocity values of the Agent, decide which is the best Animation to play at that moment, and use HALCA to do it.

3.3.2. Animation Controller

The Animation Controller needs to implement a method `animate(ref Agent a, ref CharacterRepresentation cr)`. Although the same Animation Controller could be used by different Agents and/or different Character Representations, it has sense to assume that the different Character Representations of the same Avatar could share the extracted information from the Animation Set of the main Character Representation. An avatar should walk or run at the same speed when it is represented by a 3D mesh as when it is by an animated impostor. So even if the Agent Controller is linked to the Agents, we can think that there should be at least one Animation Controller instance for each Avatar. And therefore a method `processAnimations(ref AnimationSet)` should be implemented too (although it can be empty if it is not required by your controller implementation). With this interface the user should be able to implement his own animation controllers, to have different locomotion styles, idle behaviors, etc.

3.3.3. Instancing and Palette Skinning

At every frame agents are sorted in the Scene Tree, being grouped by Avatar and by Character Representation. This way we can fill a Vertex Buffer Object with the transform matrices of all the Agent instances, send it to the GPU, and perform just one render call with instancing. If we want to animate all the instances individually, that is, with each agent having its unique animation pose, their animation information must be sent too. This could become a bottleneck and therefore a problem, when the amount of agents is too high and that information is too big.

For example, skeletal animation requires to send matrices or quaternions for each joint of every agent. When talking about thousands of agents with character representations of around 50 bones or more, the amount of information to update and send to the GPU can be very large, and thus the bandwidth between CPU and GPU can become a major bottleneck. A solution is to have all the animations loaded in the GPU and perform there the matrix palette skinning [Dud07a], having a different pose for every instance.

We therefore suggest that an Animation Set class of a Character Representation, in addition to have all the analyzed animations, implements a function `createBufferTexture()` to create the buffer texture where all the animations will be encoded. This way the buffer will be binded and the vertex shader will be able to use it to perform instancing and palette skinning at the same time.

The advantage of this will be to have individual agents playing different animations. The counterpart will be that the vertex shader will be in charge of computing the blended pose (given different animation instances and weights), by blending within two key frames for each animation, and between the resulting poses of the different weighted animations. The number of animations blended at the same time, as well as the geometry complexity of the character will have a high impact over the performance. We therefore suggest to apply this technique for representations as simple as possible, such as impostors.

3.4. Integration

Agents within a Crowd are represented by Avatars and their Transforms, being part of the Scene Graph. At runtime the Scene is rendered by our render engine using the Transform information of every Render Object. In order to render each Avatar, it takes the position and orientation of the corresponding agent and builds the transformation matrix, which is then used to render the selected Character Representation. Depending on the distance to the camera, the Avatar selects one Character Representation or another and its Shader Program (implementing the Level of Detail selection). The user sets the LODs through an interface. The simulation is carried out on another thread, executing it for every instance of every agent. Since an Agent Controller has access to the Scene and to the Crowd, it can also have access to the possible obstacles of the scene as well as to the other agents. The Animation is performed right before rendering the Avatar.

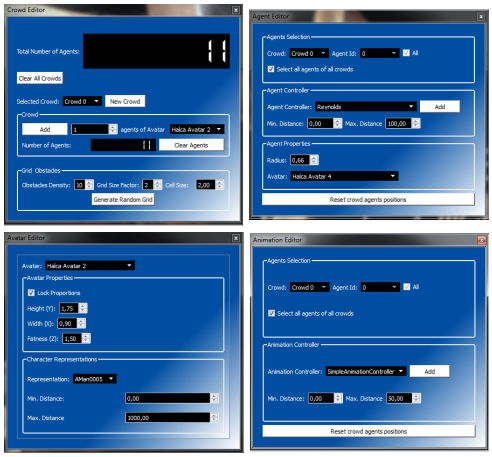

Figure 8: *GUIs for managing the crowd agents, avatars and controllers in CAVAST.*

4. Features

Although this paper presents an ongoing work to build a tool for crowd simulation, animation and rendering, we offer in the current version a fully working framework with the main functionalities and controllers already integrated. This tool is built using OpenGL 4.3 for rendering and we offer different GUIs using Qt5 (see figure 8). It is therefore straight forward for the user to create a crowd, add agents with different Avatars, and assign controllers. A Shader Manager and a Shader Editor are available allowing quick shader editing. Managers and graphic editors to configure the Levels-Of-Detail for Simulation, Animation and Visualization are also available. The user can interactively edit an Avatar by changing the different Character Representation switching distances and its main dimensions, or you can add more than one Agent Controller and more than one Animation Controller with different distance thresholds to one or more Agents for each one (please see the accompanying video for a demo).

The current tool has implemented a Character Representation called Model 3DS Representation without animations which just uses a 3ds mesh file as a character, and one HALCA Character which loads characters animated with HALCA [Spa]. In future versions we want to add a representation for characters in the FBX format by using the FBX SDK [Autb]. There are also two simple unoptimized Agent Controllers that perform a wandering behavior, one with collision detection and avoidance, and the other without it. To detect collisions each agent simply iterates over the other agents, predicts future positions (according to the current velocity) and checks intersection between the two agent radius. We also provide a simple version of some of Reynolds steering behaviors [Rey99]. In addition to that we offer a simple random grid generator and an A* Pathfinder that can work with the Reynolds controller.

Also integrated in the current systems, there is a basic Animation Controller that works with our HALCA Characters and analyzes their animations. It extracts the root speeds of each animation and therefore is able to select the best one and adjust its speed to match the speed of the agent in the crowd simulation. This way we reduce the foot-sliding effect.

Figure 9 shows a crowd of agents represented by HALCA Characters, moved by our Reynolds Controller and animated with our basic Animation Controller.

Frustum culling using bounding spheres, and occlusion culling with bouding boxes are implemented. As it has been previously mentioned, the system is also prepared to support instancing [Dud07b]. Stereoscopic visualization is also implemented, so the port to a virtual reality environment is also possible.

5. Results and Discussion

Even in its current preliminary stage, we believe that CAVAST can be a powerful tool for researchers and students. To show the potential of CAVAST, we provide some performance measurements such as frame rates for different scenes and different conditions, but it is important to notice that CAVAST is designed to be flexible and adapt to the needs and conditions of the work carried out by the user. Therefore performance measurements will strongly vary depending on the different controllers used or implemented, but having said this, we believe that the current framework provides higher performance benefits when it comes to crowd simulation, than other tools mentioned in the related work section of the paper. For example, using our test equipment (PC Intel Core i7-2600K CPU 3.40 GHz, 16 GB Ram, and a GeForce GTX 560 Ti), we can render a thousand characters represented with a non-animated 3d model of 3000 polygons each at 90 fps. If we add an unoptimized Agent

Figure 9: *A real time visualization of a crowd of agents represented by HALCA Characters. using the Reynolds Agent Controller and the basic Animation Controller from CAVAST.*

controller with collision detection, frame rate drops to 60 fps. But playing with the different LODs of the Agent controllers and of the Character Representations, we can again reach frame rates over 100 fps. Using animated characters with HALCA of 5000 polygons each, but without an Agent Controller and all playing the same animation and the same pose (thus sharing the animation data), you can have a thousand of them at 30 fps. But using an Animation Controller and giving individuality to the animation of each agent, can drop the frame rate to 23 fps. Adding an Agent Controller and thus forcing to blend more than one animation for each agent can drop again the frame rate to 14 fps.

These examples are just to illustrate that the performance of CAVAST is strongly related to the Character Representations you use, the Controllers you implement and the LOD configurations you choose. One interesting feature we would like to add is a profiling tool that could give you automatically information about your controllers and representations. This should allow the user to identify potential bottlenecks easily. It could also compare the performance between all controllers, and automatically seek for the optimum LOD set-up in order to keep a real-time frame rate. Figure 10 shows a screenshot of CAVAST with a 350 agents scenario configured to have 60 fps.

6. Conclusions and Future Work

We have presented a new prototyping and development tool for crowds research integrating animation, visualization and simulation: CAVAST. By implementing some controller interfaces and/or using some default ones, the CAVAST framework allows the user to start a new research project on crowds with all these parts running in real-time, featuring configurable Level-of-Detail and multithread. Although it may seem CAVAST do not outperforms other existing systems, our performance is strongly dependent on what you do with it. Our main contribution with CAVAST is therefore a flexible framework and a powerful tool with out-of-the-box crowd sandbox features.

As future work we want to include more features and implement more state of the art simulation and animation controllers to be provided by default. We plan to add a module for impostors generation which will allow the user to have impostors rendering for any given character. We could provide callbacks for potential simulation events (such as a fire alarm) to the Agent Controller interface, and add the corresponding trigger button to the GUI, letting the user to implement the behavior in its own controllers. We want to separate the path-finding simulation part and move it to another kind of controller interface, and link it to a navigation mesh generator module. A nice thing to add would be a generalized random generator of scenes (not just working with a grid or axis aligned objects), as well as some challenging example and benchmark scenarios. Finally, we would like to have evaluation and profiling tools, such as an automatic output of statistics or automatic perception tests. For example an automatic render of the same scene or simulation with different controllers. This might imply to add the possibility of recording and playing back simulations.

Currently the code is not multiplatform because we are using a Windows version of the HALCA library, although it should be possible to port it to other platforms when using other animation libraries. Another limitation of the current version is that the user needs to rebuild the entire application to add new controllers, so we plan for a plugin API or a scripting interface using LUA or C#. Also, characters must be added into the code before being imported, but it should be easy to add code in order to automatically import resources in a specific folder. We would like to release a free open source version of the code and make it available online soon. We believe that CAVAST could be useful also as an education tool for crowd simulation courses, allowing the students to quickly and easily visualize the results of their different algorithms.

Acknowledgements

This work has been funded by the Spanish Ministry of Science and Innovation under Grant TIN2010-20590-C02-01. A. Beacco is also supported by the grant FPUAP2009-2195 (Spanish Ministry of Education).

References

[All10] ALLBECK J.: Carosa: A tool for authoring npcs. In *Proceedings of the Third International Conference on Motion in Games* (Berlin, Heidelberg, 2010), MIG'10, Springer-Verlag, pp. 182–193. 3

[Auta] AUTODESK: 3d studio max. http://www.autodesk.com/products/autodesk-3ds-max/overview. 2

[Autb] AUTODESK: Fbx sdk. http://www.autodesk.com/products/fbx/overview. 2, 8

[Autc] AUTODESK: Maya. http://www.autodesk.com/products/autodesk-maya/overview. 2

[BAPS12] BEACCO A., ANDÚJAR C., PELECHANO N., SPANLANG B.: Efficient rendering of animated characters through optimized per-joint impostors. *Journal of Computer Animation and Virtual Worlds 23*, 2 (2012), 33–47. 7

[BF] BLENDER-FOUNDATION: Blender. http://www.blender.org/. 2

[But] BUTTERFIELD J.: Animadead: A skeletal animation library. http://animadead.sourceforge.net/. 2

[Cal] CAL3D: 3d character animation library. http://home.gna.org/cal3d/. 2, 6

[CMU] CARNEGIE-MELLON-UNIVERSITY: Panda 3d. https://www.panda3d.org/. 3

[DP85] DECHTER R., PEARL J.: Generalized best-first search strategies and the optimality of a*. *J. ACM 32*, 3 (1985), 505–536. 4

[Dud07a] DUDASH B.: Animated crowd rendering. In *GPU Gems 3* (2007), pp. 39–52. 7

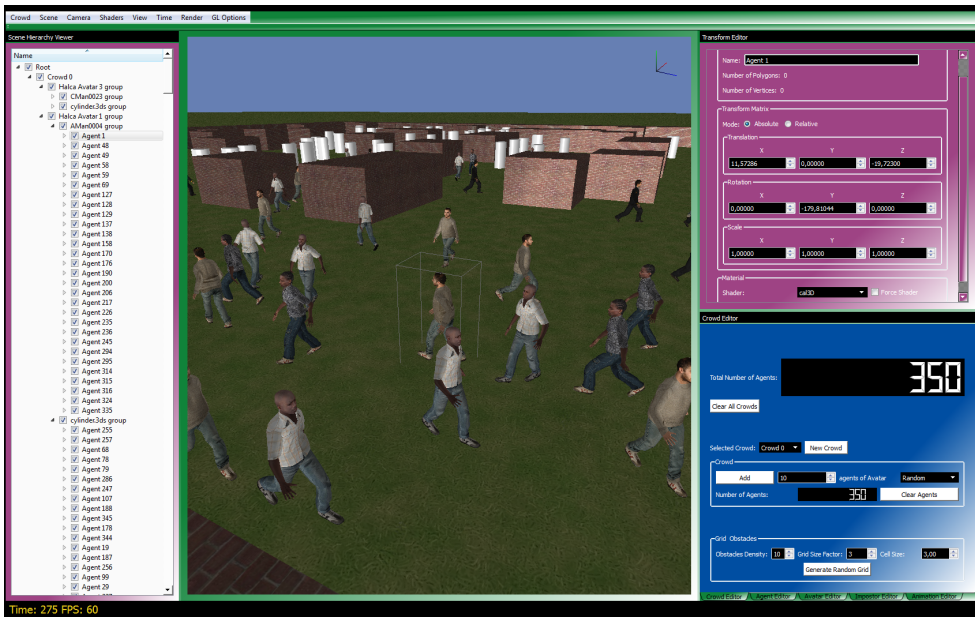

Figure 10: *Screenshot of CAVAST.*

[Dud07b] DUDASH B.: Skinned instancing. In *NVIDIA SDK 10* (2007). 5, 8

[Epi14] EPIC: Unreal engine. http://www.unrealengine.com/, 2014. 3

[Goa14] GOALEM: Golaem crowd. http://www.golaem.com/content/products/golaem-crowd/overview, 2014. 2

[GU] GAMEBASE-USA: Gamebryo. http://www.gamebryo.com/index.php. 3

[KL02] KOENIG S., LIKHACHEV M.: D* Lite. In *National Conf. on AI* (2002), AAAI, pp. 476–483. 4

[LGT03] LIKHACHEV M., GORDON G. J., THRUN S.: ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In *NIPS* (2003). 4

[Lor07] LORACH T.: Gpu blend shapes. *NVidia Whitepaper* (2007). 7

[Mas14] MASSIVE: Massive software. http://www.http://www.massivesoftware.com, 2014. 2

[MTLTM88] MAGNENAT-THALMANN N., LAPERRIRE R., THALMANN D., MONTRÉAL U. D.: Joint-dependent local deformations for hand animation and object grasping. In *In Proceedings on Graphics interface'88* (1988), pp. 26–33. 6

[OE11] O'SULLIVAN C., ENNIS C.: Metropolis: Multisensory simulation of a populated city. In *Games and Virtual Worlds for Serious Applications (VS-GAMES), 2011 Third International Conference on* (May 2011), pp. 1–7. 4

[Ogr14] OGRE: Ogre: Object-oriented graphics rendering engine. http:://www.ogre3d.org//, 2014. 3

[PAB08] PELECHANO N., ALLBECK J., BADLER N.: *Virtual Crowds: Methods, Simulation, and Control*. Morgan & Claypool, 2008. 1

[Rey] REYNOLDS C.: Opensteer: Steering behaviors for autonomous characters. http://opensteer.sourceforge.net/. 2

[Rey99] REYNOLDS C.: Steering behaviors for autonomous characters. In *Game Developers Conference* (1999). 5, 8

[RGT] RAD-GAME-TOOLS: Granny 3d. http://www.radgametools.com/granny.html. 2

[Sha] SHAPIRO A.: Smartbody, university of southern california institute for creative technologies. http://smartbody.ict.usc.edu/. 4

[SKFR09] SINGH S., KAPADIA M., FALOUTSOS P., REINMAN G.: An open framework for developing, evaluating, and sharing steering algorithms. In *Proceedings of the 2nd International Workshop on Motion in Games* (Berlin, Heidelberg, 2009), MIG '09, Springer-Verlag, pp. 158–169. 2

[SMKB13] SHOULSON A., MARSHAK N., KAPADIA M., BADLER N.: ADAPT : The Agent Development and Prototyping Testbed. *ACM SIGGRAPH I3D* (2013). 3

[Spa] SPANLANG B.: Hardware accelerated library for character animation (halca). http://www.lsi.upc.edu/∼bspanlang/animation/avatarslib/doc/. 2, 6, 7, 8

[TLC02] TECCHIA F., LOSCOS C., CHRYSANTHOU Y.: Image-based crowd rendering. *IEEE Comput. Graph. Appl. 22*, 2 (2002), 36–43. 7

[TM13] THALMANN D., MUSSE S.: *Crowd Simulation, Second Edition*. Springer, 2013. 1

[UCT04] ULICNY B., CIECHOMSKI P. D. H., THALMANN D.: Crowdbrush: interactive authoring of real-time crowd scenes. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2004), SCA '04, Eurographics Association, pp. 243–252. 6

[Uni14] UNITY: Unity: Game engine. http:://unity3d.com/, 2014. 3