Creating a Virtual Audience for the Heritage of Ancient Theaters and Odea

D. Thalmann, R. Cetre, B.Ulicny, P. de Heras Ciechomski, M. Clavien
EPFL, VRLab, CH-1015, Lausanne, Switzerland

Abstract. The objectives of this work are a recreation of 3D models of selected archaeological open-air and roofed theatres of the Hellenistic and Roman period, and a creation of the real-time crowd engine in order to animate autonomously the virtual audience. We present a methodology for creation of virtual audience using a collection of tools, including custom ones developed specifically for this task and standard 3D modeling application.

1. Introduction

The aim of this paper is to describe our work on the crowd animation and control module. This module is used to create a real-time interactive scenario of virtual audience populating reconstruction of an ancient odeon.

For the ERATO[1] project [1] we target to handle an order of magnitude larger crowd than in previous works on crowds in virtual heritage [2]. This brings new challenges, both for a design of the real-time animation and control software, and for scenario content creation and management.

We developed a rule based crowd behavior engine which handles reactions of the audience to various events, and as well alleviates variety management of animations.

For creation of scenarios we developed a tool named CrowdBrush which allows to distribute and to modify crowd members in a scene in intuitive manner using metaphors of artistic tools [3].

2. Methodology for creation of virtual audience

Our goal is to create a set of human models that would be light enough to allow for rendering of up to thousand of virtual humans simultaneously present on the screen at interactive framerates.

One of the biggest challenges is how to create variety while keeping the speed of rendering and memory requirements at acceptable levels. The basic idea is to create several template meshes (see Figure 2), which are then in run-time cloned to produce large number of people and modified by applying different textures, colors and scaling factors to create a variety.

For the modeling of the virtual humans we used the commercial software package 3ds max [4] (see Figure 1). We wrote a mesh and skeleton exporter using MaxScript. The output of the exporter is a data directly usable by our real-time rendering and animation engine (see Appendix for more details).

---

[1] Sponsored by the Swiss Federal Office for Education and Science.

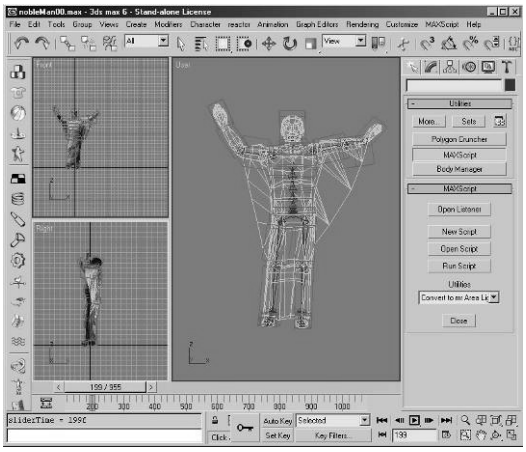

Figure 1: *Creation of virtual human in 3ds max.*

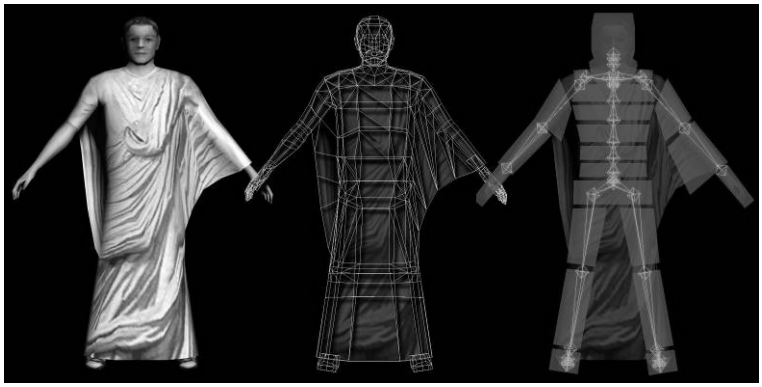

Figure 2: *Virtual human template (full model, mesh, skeleton).*

The mesh itself consists of vertices and triangles. The exported triangles have associated information such as texture coordinates, materials and skeleton bindings. Skeleton bindings specify weights of the influences of bones for vertices of the mesh. We use Bones Pro 3 plug-in [5] for defining these bindings. Key deformation postures have to be tested and skinning weights have to be adjusted to produce correct mesh forms (see Figure 3).

Figure 3: *Key deformation postures.*

The next step in the data-production pipeline is exporting a reference mesh and a reference posture describing the initial animation for applying skeletal transforms using the Bones Pro 3 plug-in. This allows exporting of any animation for a particular skeleton. The animation is defined as a list of bones in the hierarchy with translation and rotation for every sampled keyframe.

An important part of the data production pipeline is a creation of textures. Textures add details to models and allow reusing template meshes to create variety of audience. Concerning the performance of the rendering engine, we had to take into account organization of the textures. The best results are achieved by using a single texture per human which allows sorting models by their materials.

Using methods outlined above we created several meshes for different social classes (see Figure 5).

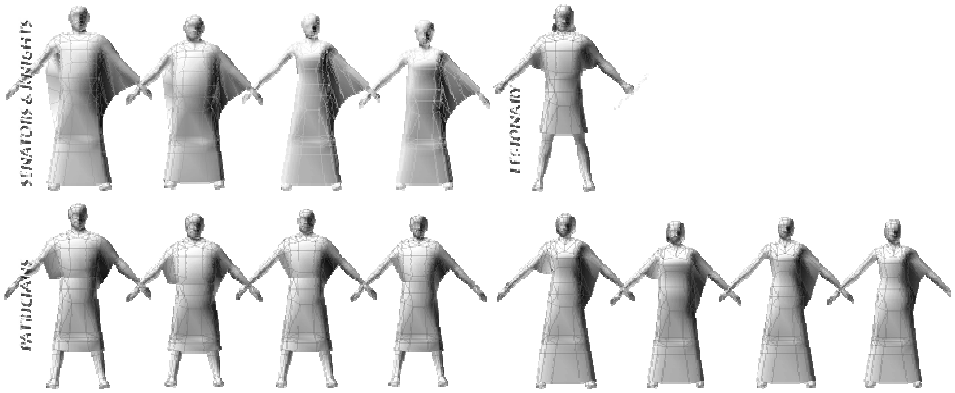

Figure 5: *Mesh design.*

After cloning and applying different textures, we achieved higher levels of variety at relatively low costs considering the run-time performance (see Figure 6).

Figure 6: *Varied crowd created by texture combinations.*

After designing of a set of templates, we created a large audience by cloning in run-time using our crowd rendering and animation engine. Finally, we integrated the virtual audience into the model of the Aphrodisias odeon (see Figure 7).

Figure 7: *Integration of audience into model of Aphrodisias odeon.*

Following figures shows samples of images of our actual set of virtual Romans and an example of audience animations:

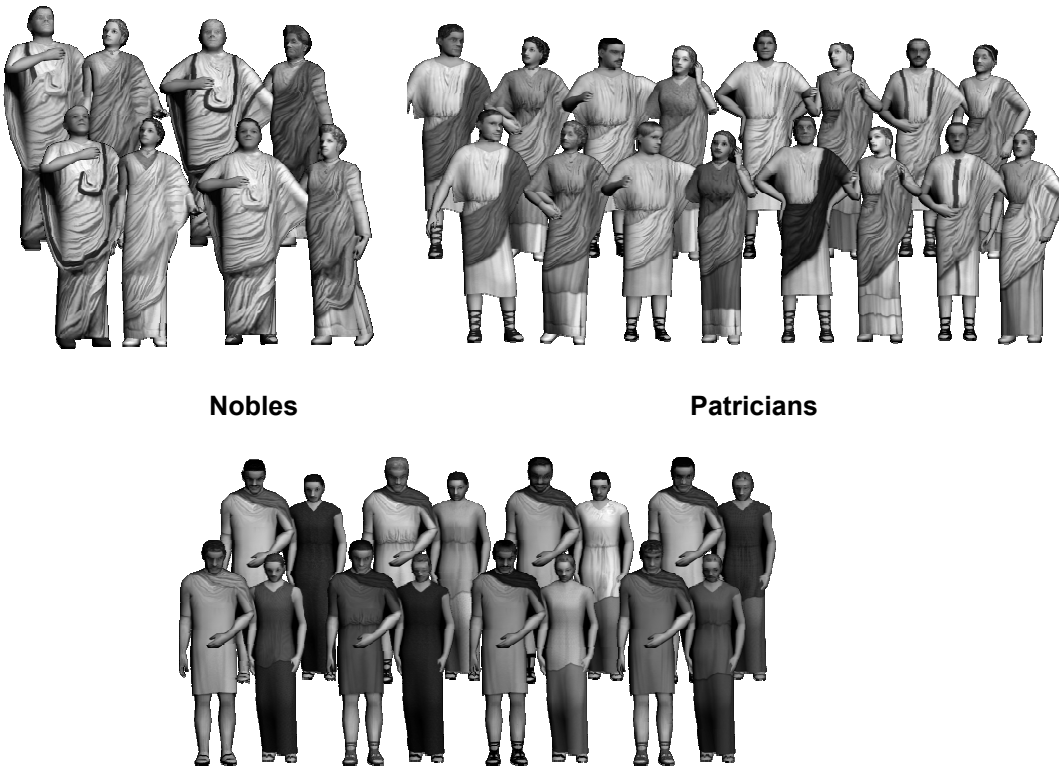

Figure 8: *Virtual Romans.*

2.1 Designer workflow

The designer exports the mesh, skeleton binding, texture combinations and so forth, which can be then directly deployed in the real-time environment. The designer uses two applications: the first one is a quick test application called VRSkin for tuning of performance and level-of-detail distances (see Figure 6), the other one being a full ERATO application built using VHD++ platform [6] (see Figure 7). In VRSkin application, a number of templates can be instantly visualized and animated configured by a simple text initialization file allowing for a fast turnaround. After being tried in VRSkin, models are put in the ERATO application for further integration with architectural model.

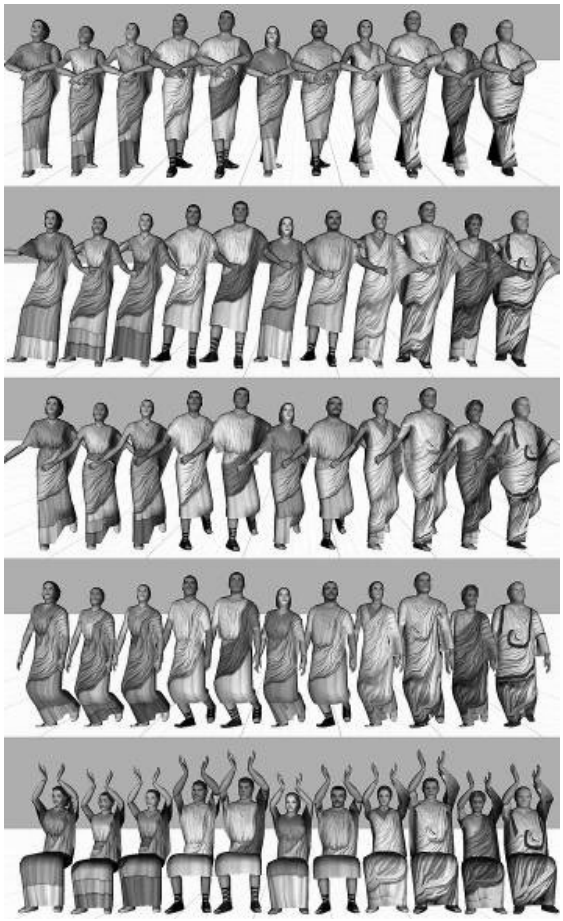

Figure 9. *Examples of animation.*

All our humans have different meshes, for example, a plebeian man uses a different mesh than a noble man (see Figure 8). However, they all use the same skeleton, so that any animation that is applied to this skeleton can be shared by all template humans (see Figure 9). This allows the designer to export the animation only once for all templates, which is especially important when increasing number of templates.

3. Crowd behavior engine

We implemented a rule based behavior engine [7] and a simple spatial displacement of the humans to handle more autonomous characters. We used a simple reactive rules system as we needed fast execution for up to several thousands of agents.

3.1. Behavioral rules

At the core of the behavior engine are behavioral rules. Behavior rules react to internal and external events depending on the state of the agent, triggering sequences of actions such as displacement or animations. Rules are composed of three parts:

1. **Selection part** - defining which agents will be affected by the rule.
2. **Condition part -** defining when the rule is applicable.
3. **Consequent part** – defining what will be the result of the rule firing.

```
FOR ALL
WHEN EVENT = action_neutral_cycle
THEN
  SCRIPT
    WAIT RANGE 3.00 5.00
    PERFORM_ACTION RANDOM animation_neutral ACCORDING_TO body
    SEND_EVENT action_neutral_cycle TO SELF
```

Table 1: *Example of behavioral rule.*

Behavioral rules have two main functions. The primary one is orchestration of the scenario, where the rules tell in every moment for every involved character what to do. The secondary role is to help with management of variety. Several mechanisms for handling the variety are available:

- Animations can be played with varied speed selected from some random range.
- Triggering of the animation can be delayed by random amount of the time.
- Higher-level actions selected by the rules can be executed in different manner, using selection from predefined sets of animations with the same semantics and grouped according to modifying attribute (such as gender, or social class).

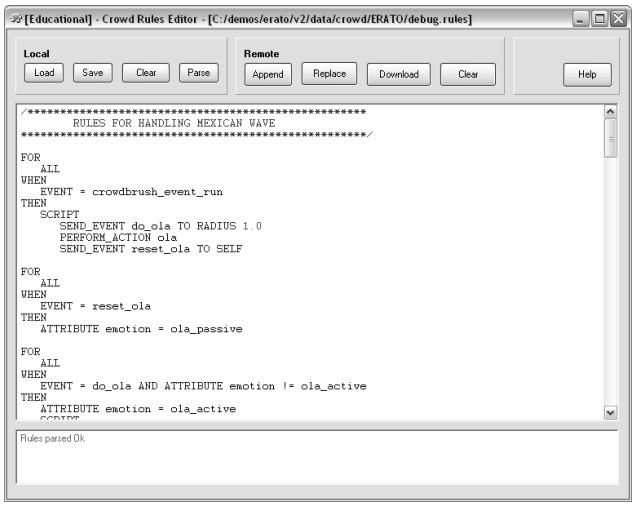

Figure 12: *Crowd rules editor.*

The behavioral rules are editable in run-time using rules editor connected with the behavior engine (see Figure 12).

3.2. Motion system

Displacement of the agents around the environment is at the lowest level of the behavior computation, providing action execution for the behavioral rule system. Agents can move around the environment following paths while avoiding collisions with other agents. Positions of the agents between waypoints are interpolated taking into account their desired speed and actual simulation update rate. Along the way, agents play looping keyframed animations of different walking styles or running depending on the displacement speed.

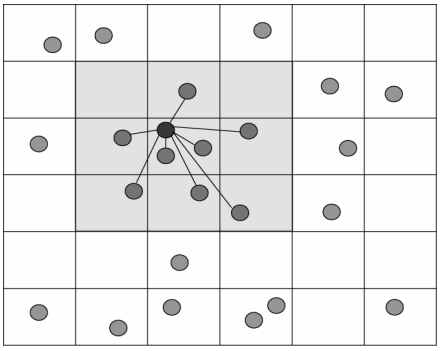

Figure 13: *Bin-lattice optimized collision avoidance.*

Collision avoidance is computed by adding displacements to agents' positions due to proximity of other agents using a social force model [8]. In order to improve efficiency of the collision avoidance, spatial queries need to be optimized; otherwise collision detection would become a performance bottleneck much sooner than rendering. We use a simple bin-lattice space subdivision to minimize the number of collision queries [9]. In this scheme, agents in every update frame are distributed into an uniformly spaced grid, and then collision queries are performed only for the agents that are in the same or neighboring cell (see Figure 13).

4. Crowd scenario authoring

We developed a custom tool helping with authoring of the audience scenario called CrowdBrush. CrowdBrush tool allows to distribute agents in the virtual environment and then to apply a set of features or behaviors to these agents. The designer manipulates a set of virtual tools affecting the objects in a three-dimensional space. Different tools have different visualizations and perform different effects on the scene including creation and deletion of crowd members, changing of their appearances, triggering of various animations, setting of higher-level behavioral parameters, setting waypoints for displacement of the crowd, or sending of events to a behavior subsystem.

Figure 14 shows examples of brushes used to create or change an agent to a particular social class and brushes that will alter an emotional state of the agent. A more detailed description can be found in [3].

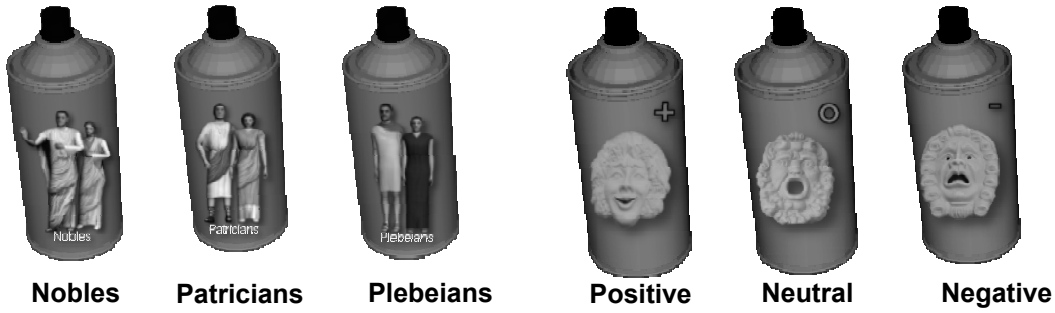

Figure 14: *Virtual tools for changing a social class (left) and an emotional state of the agent (right).*

5. Interactive scenario

We are preparing interactive scenario which will present the user with recreation of theatre atmosphere in ancient Roman times. The user will observe the audience and have influence on their behaviors using a graphical user interface. Following figure shows an outline of the scenario flow:

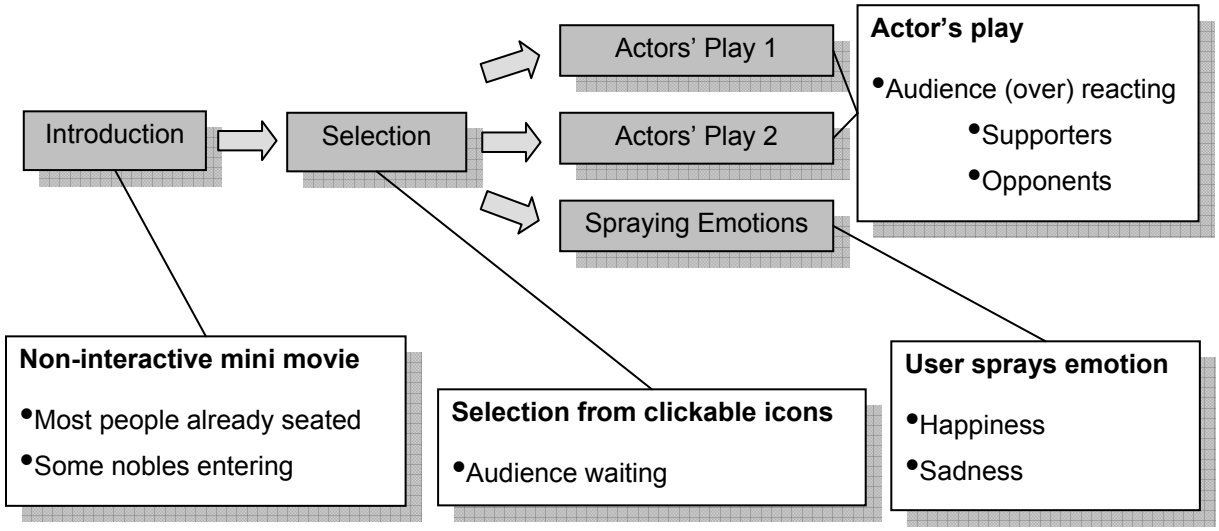

Figure 16: *Interactive scenario outline.*

The scenario is taking inspiration from the historical literature on audiences in Roman times. We extracted information on Roman behavior in theatre from "Das Theater Publikum der Antike" by Heinz Kindermann [10]:

- People entered the odeon according to their social status; lower classes came first (upper and farther seats); most renowned senators or nobles arrived last, and the audience made an ovation for them (standing applause, shouting his name and salute in unison were the most usual ovations).
- Roman audiences were very noisy: chatting, flirting, moving around, etc. and it was very difficult to get them calm down to start the show (often theatre police had to intervene).
- Nobody was allowed to leave the odeon during the show.
- Famous actors were treated as superstars; they had their own "fan-club"; and as their popularity was measured by their fan's support, they often paid people in the audience to acclaim them and humiliate their rivals.

- Positive reactions: clapping hands, shouting “da capo!” or “bravo!”, unison clapping
- Negative reactions: hissing, feet rattling, shouting etc.
- Fans were over-reacting to such extent that sometimes the hissing could last until the actor left the stage.
- The audience was not only reacting to their favorite actors, but they were reacting to the play as well: approving, disapproving, laughing loud, crying, hailing a character, discussing with a neighbor, and so on.

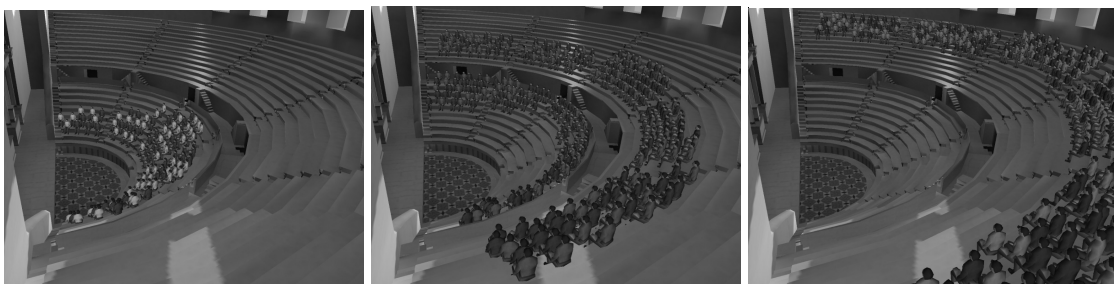

Figure 17: *Distribution of the audience in odeon: a) nobles, b) patricians, c) plebeians.*

The important part of the scenario is a distribution of the audience in the odeon. We created plan of the distribution according to historical sources (see previous section) and then used the CrowdBrush tool to distribute agents of different social classes in the building (see Figure 17).

References

[1] ERATO Project website, http://www.at.oersted.dtu.dk/~erato

[2] B. Ulicny, D. Thalmann: Crowd simulation for virtual heritage, *Proc. First International Workshop on 3D Virtual Heritage*. Geneva, pp.28-32, 2002.

[3] B. Ulicny, P. de Heras, D. Thalmann, “CrowdBrush: Interactive Authoring of Real-time Crowd Scenes”, *Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation ‘04*, 2004

[4] 3ds max, 2004. http://www.discreet.com/3dsmax

[5] Bones Pro 3, 2004, http://www.digimation.com

[6] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, D. Thalmann, “VHD++ Development Framework: Towards Extendible, Component Based VR/AR Simulation Engine Featuring Advanced Virtual Character Technologies”, *Proc. Computer Graphics International (CGI)*, 2003

[7] B. Ulicny, D. Thalmann: Towards interactive real-time crowd behavior simulation. *Computer Graphics Forum* 21, 4 (Dec. 2002), 767–775.

[8] D. Helbing and P. Molnar, “Social force model for pedestrian dynamics”, Phys. Rev. E , Vol. 51 , pp. 4282-4286 , 1995.

[9] C. W. Reynolds: Interaction with groups of autonomous characters. In *Proc. Game Developers Conference ’00* (2000), pp. 449–460.

[10] H. Kindermann: *Das Theater Publikum der Antike,* Otto Müller, Salzburg, 1979.

Appendix

When exporting humans, we needed to address the issue of a different coordinate system in 3ds max and OpenGL used to handle rendering in our real-time application. In 3ds max, the basic axes are aligned as follows: z up, y pointing into the screen and x is pointing to the right. On the other hand, OpenGL has z out of the screen and y is up (see Figure 17). This results in humans facing the ground if not taken care of properly.

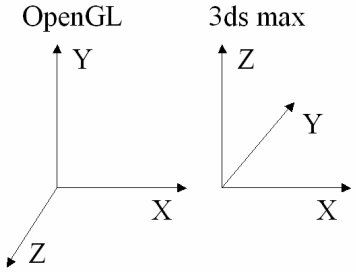

Figure 17: *Coordinate systems in OpenGL and 3ds max.*

The simple solution is to rotate by minus 90 degrees around the x axis to align the two systems. By keeping all vertices and transforms as they were from Max and then adjusting with a rotation the resulting human was properly aligned. Given the rotation around the x axis is R(-90), the local transform in Max is T_{max} and the vertex in Max is V_{max} then after loading the mesh and animations the resulting vertex in OpenGL becomes $V_{ogl} = R(-90)*T_{max}*V_{max}$. This solution works as long as you do not interact with the animations exported from Max or just use the animations exported from Max not trying to load any others from publicly available databases.

A more complex solution is needed when we have to interact with the bone transforms in their local coordinate system, for example while using more complex motion generators beyond simple keyframing. At that point we have to interact with the individual bones of characters and the simple solution from above will no longer work. We have to replace the transforms from 3ds max with $T_{ogl} = R(-90)*T_{max}*R(90)$ and replace the vertices with $V_{ogl} = R(-90)*V_{max}$, which in the end will be the same globally as the simple solution and will now have a correct local transform per bone.

$R(-90)*T_{max}*R(90) * R(-90)*V_{max} == R(-90)*T_{max}*V_{max}$

In row major format:

R(-90) = [1 0 0 0; 0 0 1 0; 0 -1 0 0; 0 0 0 1]
R(90) = [1 0 0 0; 0 0 -1 0; 0 1 0 0; 0 0 0 1]

A local transform:

T_{max} = [Rotation Translation] = [R_{xx} R_{yx} R_{zx} T_x; R_{xy} R_{yy} R_{zy} T_y; R_{xz} R_{yz} R_{zz} T_z; 0 0 0 1]