The 6th International Symposium on Virtual Reality, Archaeology and Cultural Heritage
VAST (2005)
M. Mudge, N. Ryan, R. Scopigno (Editors)

Real-time Shader Rendering for Crowds in Virtual Heritage

Pablo de Heras Ciechomski, Sébastien Schertenleib, Jonathan Maïm, Damien Maupu and Daniel Thalmann

VRLab, EPFL
CH-1015, Lausanne, Switzerland
{pablo.deheras, sebastien.schertenleib, jonathan.maim, damien.maupu, daniel.thalmann}@epfl.ch
http://vrlab.epfl.ch

Abstract

We present a method of fully dynamically rendered virtual humans with variety in color, animation and appearance. This is achieved by using vertex and fragment shaders programmed in the OpenGL shading language (GLSL). We then compare our results with a fixed function pipeline based approach. We also show a color variety creation GUI using HSB color space restriction. An improved version of the LOD pipeline for our virtual characters is presented. With these new techniques, we are able to use a full dynamic animation range in the crowd populating the Aphrodisias odeon (which is part of the ERATO project), i.e., a greater repertoire of animations, smooth transitions and more variety and speed. We show how a multi-view of the rendering data can ensure good batching of rendering primitives and comfortable constant time access.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Virtual Heritage]: Computer Graphics Animation

1. Introduction

We are working on the project - Identification, Evaluation and Revival of the Acoustical heritage of ancient Theaters and Odea - (ERATO) [ERA05], where our responsibility lies in recreating the Aphrodisias odeon in Turkey and filling it with a virtual humans audience that follows a play on stage [dHCUDC04, UdHCT04, dHCSMT05, TCU*04]. The work on the Odeon is now finalized and some pictures of it can be seen in Figure 1.

Since the beginning of the project, our focus has always been on creating a graphical pipeline that could be applied to many different scenarios and cultural heritage settings. We would like the user of our library to be able to quickly get a heritage application running from scratch, through the use of customizable meshes, textures, colorings, behaviors and scenarios. One of our contributions in this paper lies in the use of HSB color constraining with a tool specifically developed for our designer needs. These color constraints are specifically designed for fast prototyping and final delivery of virtual heritage crowds with historically significant constrained color ranges. We are able to display a full dynamic range of animations and transitions between animations computed with a minimal memory footprint. This is achieved on the fly for an entire crowd of virtual characters, thanks to an improved deformation pipeline using hardware vertex and fragment shaders. Finally we show how a tool from the OGRE3D distribution [Ogr05] can be adapted and improved to handle virtual human LODs with texture seam preservation.

Variety is defined as having different forms or types and is necessary to create believable and reliable crowds in opposition to uniform crowds. For a human crowd, variation can come from the following aspects: gender, age, morphology, head, kind of clothes, color of clothes and behaviors. We propose here a way to create as many color variations as possible from a single texture. Several works have been done on the subject. [TLC02] propose an image-based rendering approach to display a crowd. Variety comes from multipass rendering where different colors are assigned to significant parts of the body such as clothes, hair and

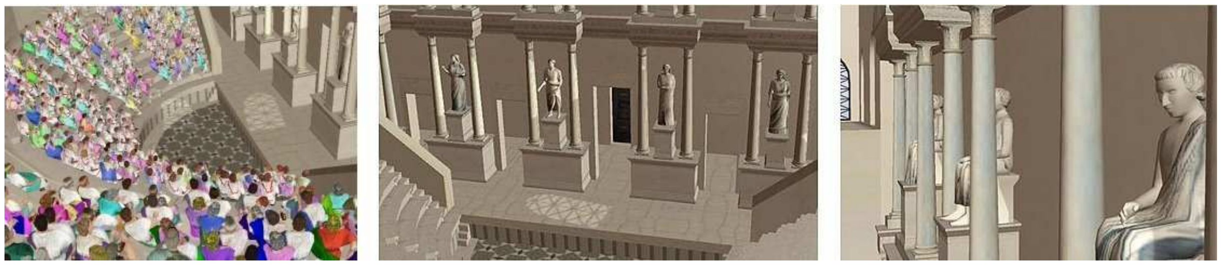


Figure 1: *The final Aphrodisias odeon version with sculptures*

skin color. [dHCUDC04] propose to create several template meshes, which are at run-time cloned to produce a large number of people. These clones can then be modified by applying different textures to create variety. Similar work that also treats meshes with variety uses a static pre-deformed mesh ([RFD04, DHOO05, CLM05]). The difference between these approaches is the use of one, two, or three levels of geometrical details before starting to render the impostors. There is one exception to the rule of going from strict geometric rendering to impostors in [WS02], where they use a smooth transition from triangle mesh-based rendering to particle splats. However, none of these approaches attack the problem of editing colors variety according to designer needs, nor do they address the problem of using a large animation palette. Indeed, all crowd rendering papers to date employ only one walking animation except for [UdHCT04, dHCUDC04, dHCSMT05, GSM04]. The main reason for this lack of animation variety is that when using a billboarded crowd, this variety needs to be constrained since billboarded approaches could be extended to use more animations but would explode the memory requirements in the process. Having one or a few animations is fine when one works with static meshes, but as can be seen in [dHCSMT05], this leads to a memory explosion when several template meshes are added with multiple animations each. This was solved in [dHCSMT05] by having dynamically deformed meshes with animations and geometrical caches. We build our meshes upon this idea and improve the color editing, the animation variety and the overall performance.

2. Variety Rendering

Our variety in rendering extends the way Tecchia *et al.* [TLC02] use to create color variety from a single texture to dynamically animated 3D virtual humans. Our goal is to have a wide variety of colors and appearances by combining our texture and color variety (see Section 5). Each mesh has a set of interchangeable textures and the alpha-channel of each texture is segmented in several zones: one for each body part. This segmentation is done using a desktop publishing software (in our case Adobe PhotoShop [Pho05]). We present two possible approaches to use the alpha layer

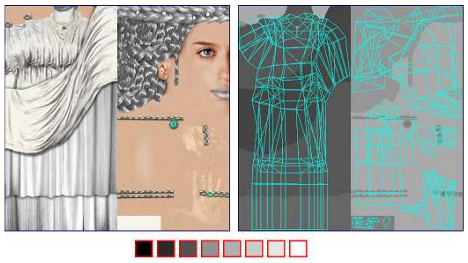


Figure 2: *Texture and alpha zone map of a patrician woman*

information for creating color variety. The first approach is software-based and runs on graphics cards with a fixed function pipeline. The second one uses a fragment shader.

2.1. A Software Approach: Fixed Function Pipeline

An alpha zone is denoting a part of the texture of the character that is to be modulated with a certain color. For example an alpha zone is a specific part of a dress or some jewelry that we want to be able to color to differentiate crowd members from each other. Some triangles, denoted as "dirty" triangles, can span several such alpha zones that have to be colored differently. An example of such triangles can be seen in Figure 2 where triangles close to the border between the face skin color and the hair are overlapping. A solution consists in re-triangulating "dirty" triangles to have them cover each zone with more triangles. Even if this could be easily achieved, it is to be rejected, because there would be no control on the amount of new triangles generated and a nonsensical situation could appear: a lower LOD mesh having more triangles then a higher LOD mesh. Our approach consists of using alpha tests on a split mesh to reduce the work of the graphics card. Meshes are quickly split at startup in several sub-meshes: one per group of uniform triangles, plus one for the "dirty" triangles. This splitting is achieved by plotting every triangle over the texture's alpha layer. During the plotting, if a change in the alpha value appears, the triangle is then considered as "dirty". Once the mesh is split, multipass rendering is done only on the "dirty" triangle sub-mesh.

The main drawback of the software approach is that the rendering complexity depends on the amount of "dirty" triangles. If some alphas overlap a large amount of triangles, the rendering will slow down accordingly, which makes the frame rate depend considerably on the chosen texture. Moreover, it also depends on the way alpha zones are designed. One willing to use this splitting mesh technique should take into account this issue and try to reduce as much as possible the amount of "dirty" triangles by being extra careful when creating textures. A way to ensure this is to have zones with uniform alpha values connected by a large pixel neighborhood. However, this approach is able to run on our crowd generator on a large install-base of machines.

2.2. A Hardware Approach: Shaders for Color Variety

In our continuing process to improve the execution speed of our crowd inside the Aphrodisias odeon, we have decided to explore the performance of high-end consumer graphics cards, specifically the nVidia Geforce 6 series [nVi05] with support for GL Slang (GLSL) by 3DLabs [3DL05] and shader model 3.0. Each character is able to have up to 256 different alpha key areas corresponding to a certain body part like in [GSM04]. Using the software approach, this is completely unreasonable, as it would require 256 passes for the uniform alpha triangles and another 256 passes over the dirty ones. Although the hardware approach could manage such a number of areas much better than the software approach, in practice, we only use up to 10 different areas, such as lips, eyes, hair, parts of the dress, jewelry etc. In the fragment shader, we have to determine for each pixel which area it is part of and then color it with the appropriate modulating color. In our first implementation of the fragment shader, we had an *if-else* statement check for determining the correct color to apply with a limitation of 8 alpha zones. Since Shader model 3.0 allows nested *if* statements, we can do with only 3 *if* evaluations per pixel. Another way to program the fragment shader is to send a one dimensional texture and map the alpha value to the color we want to modulate with, for a specific character. In order to batch drawing calls [Wlo03], we put all the individual one dimensional textures into one 2D texture of 1024 times 256 size. Each row in this texture atlas maps from an alpha value to a color value and only one extra variable has to be sent to the mesh in the form of an identifier for which row to sample from.

2.3. Filtering

Our color variety tool rests on a precise control of alpha key values. While uploading textures to the graphics card, such filtering as "nearest" filtering is preferable to a "linear" filtering. Indeed a "linear" filtering would create new alpha values at the border of two alpha zones and thus, pixels at these borders would not be drawn. However, nearest neighbor filtering is very gross and, to soften the texture, Mipmaps are required [Wil83]. OpenGL's Mipmap creation tool cannot be used here since we need separate bilinear and nearest neighbor filtering. In fact, the alpha layer itself has to be nearest neighbor filtered separately, while the RGB layer must be bi-linearly filtered when the Mipmaps are being built. This is not yet implemented.

2.4. Color

The color variety presented here is based on texture color modulation. Each fragment is colored by modulating the pixel color by the texture color: thus, the value produced by the texture function is given by:

$$C_v = C_t C_f \tag{1}$$

where f refers to the incoming fragment and t to the texture image. Colors C_v, C_t, and C_f are values between 0 and 1. In order to have a large panel of reachable colors, C_t should be as light as possible, *i.e.*, near to 1. Indeed, if C_t is too dark, the modulation by C_f will give only dark colors. On the other hand, if C_t is a light color, the modulation by C_f will provide not only light colors but also dark ones. This explains why part of the texture has to be reduced to a light luminance, *i.e.*, the shading information and the roughness of the material. Passing the whole texture as luminance does not make sense. First, there is no gain in memory : OpenGL will emulate an RGB texture based on luminance values, because graphics cards are optimized for RGB textures. The drawback of passing the main parts of the texture to luminance is that "funky" colors can be generated, *i.e.*, agents are dressed with colors that don't match. Some constraints have to be added when the modulating colors randomly. With the RGB color system, it is hard to constrain colors effectively. That is why we use the HSB system [Smi78], also called HSV, meaning Hue, Saturation, Brightness (respectively Value). This model is linked to the human color perception and is more user-friendly than the RGB system. In Section 5, we present a graphical user interface that has been created for helping designers to set constraints on colors.

2.5. A Hardware Approach: Shaders for Deformation

By re-writing the vertex deformation step into a hardware-based vertex shader, we can no longer re-use the result of a computation on the vertices in a software memory caching scheme, as in [dHCSMT05]. However, the benefits are twofold: first, as the graphics pipeline on our cards are made of 6 vertex processors working in parallel [nVi05], we have a major speed-up. Second, our variety coloring can now be done in one pass per pixel in the fragment shader (not as in the software approach, which is a multi-pass solution [dHCSMT05]). Thus, we have 16 of these fragment shaders working in parallel.

Each vertex needs the array of deformation matrices, indices and weights for these matrices, a normal, a position, and a texture coordinate. A way of sending this data to the graphics card is through attribute vectors [3DL05]. Instead, we prefer to send the per vertex attributes embedded in the standard components [SWND03], *i.e.*, in our case : the color, texture coordinate, position and normal. This is achieved by storing up to three matrix indices in the RGB color component, the number of affecting bones in the alpha color component, the weights of bones one through three in the w component of the position and the third and fourth component of a four dimensional texture coordinate. We also ensure that for low LODs, the character will not use more than one bone. This allows us to use the *if* statement in the vertex shader that is part of shader model 3.0 instead of activating another program. An *if* statement only costs 2 cycles according to nVidia sources.

To better use the graphics card, we need to store the geometric data, the weights, the texture coordinates, the number of bones affecting each individual vertex and so on. This is done by storing the glDrawElements call in a display list. The OpenGL driver will store it in in an optimal form so that, for rendering a character, we will only need to send a few attributes; in this case the transformations of each bone, which row to sample from the 2D alpha to color texture, and a variable stating (for a character supposed to use only one bone per vertex). Thus, we decrease the communication to the graphics card to a minimum and the rendering is achieved in one pass.

2.6. Comparison of Shaders Versus a Software Approach

In our first approach to create color variety for the virtual humans, we were using a multi-pass rendering for each colored region and we optimized it so that only the triangles that were fully inside a region were rendered in blocks. The triangle that passed several regions were rendered several times, using alpha checks (see [dHCSMT05]). With a software approach, the decision process is done by the alpha function which has to do up to 8 passes of sub-mesh rendering and one pass per triangle group fully withing an alpha region. This gives us 1 pass per sub-mesh and 8 passes for the dirty triangles, which is still faster than to use 8 passes considering all triangles as dirty. In the shader approach only one pass is done over the triangles though the fragment program has to run on all rendered pixels. Since the fragment program is only 6 GLSL lines using two texture look-ups to get the final color it is extremely fast.

On the side of the vertex shader we only do one OpenGL display list call for each human and upload the attributes of bone matrices along with only two extra variables. This gives us an enormous advantage in terms of rendering primitive calls comparing to the software approach which does up to 16 glDrawElements calls. Storing the geometry and variables of the mesh in graphics card memory even if it has to be deformed on the hardware gives a considerate boost in performance as seen in the results section.

3. System Architecture

Crowd simulations require addressing many problems from different perspectives. In our experiments, a typical scenario features more than a few hundreds distinct virtual characters. The number of assets to be created and controlled interactively within the simulation is becoming a predominant factor. Naturally, the real-time performance of such a virtual environment depends on the appropriate usage of a 3D API such as OpenGL [Ope05]. These APIs come with design particularities that oblige the developers to organize their rendering pipeline in a very strict order. Unfortunately, this organization is not intended to be easily controllable for high-level simulation designers. Very often, developers have to make some compromise between real-time performance and the flexibility offered by their system. In order to overcome this barrier, we have improved our scene management by extending the way the data can be fetched. The idea is to provide different views based on the specific requirements of every module.

3.1. Multi-View Scene Representation

By providing different view representations of the same shared data, we can optimize both the access time and the different components computation. For instance, the GPU rendering pipeline and the semantic representation have distinct requirements. The 3D rendering module needs to classify 3D meshes among their rendering cost penalty (OpenGL state sorting, shaders parameters bindings, etc), while the semantic view may access them by their unique ID. Our approach tends to overcome the limitations of classifying objects in a unique list or queue. To implement this multi-view representation, we are relying on the multi-index container from the [Boo05] library. This container maintains multiple indices with different sorting and access semantics. It goes beyond the single view representation that the STL map or set may offer. The original idea of this concept came from the indexing theory as used in relational databases. Thus, we can simultaneously optimize data access for spatial, state or semantic considerations, by fitting precisely their local requirements. Figure 3 depicts such a container with three view indices on the same shared data.

This design does not only allow a perfect balancing between the 3D rendering and semantics view, but it also offers different possibilities to optimize every single component. For instance, 3D rendering optimizations are often platform specific : some hardware may be more efficient to fetch different textures than to modify their lighting computation mode. By extending our multi-index container with additional views, we can adapt our system more easily. This is particularly important for controlling the simulation, as different semantic indexing may be necessary. They may represent more logical or intuitive views for accessing the different objects could they be using unique ID or by classifying them in different categories (human, dynamic object,

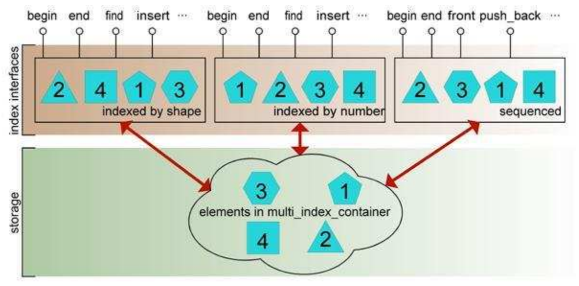

begin end find insert ···
begin end find insert ···
begin end front push_back ···
index interfaces
2 4 1 3
indexed by shape
1 2 3 4
indexed by number
2 3 1 4
sequenced
storage
3 1
elements in multi_index_container
4 2

Figure 3: *Multi-view representation*

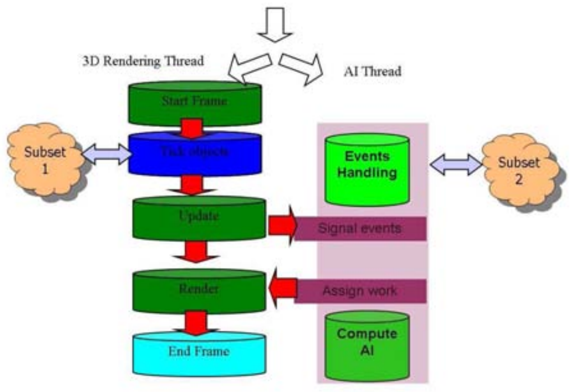


Figure 4: *Threading Work Flow Design Model*

static object). From a performance perspective, as our virtual worlds are populated with thousands of dynamic entities, it is really important that we keep a constant time access on every representation. The container implementation is ensuring this requirement. Moreover, the spatial efficiency becomes acceptable with regards to memory consumption as the number of indices grows. Finally, the memory fragmentation remains minimal especially in comparison to manual management, which can show up in more usable memory available and better performance.

3.2. Multithreading for Crowd Simulations

Recent developments in CPU technology have extended the install-base of computers capable of running multiple hardware threads simultaneously. However, taking advantage of this advance in hardware technology involves significant high level modifications within the system design. [Kru05] have shown that providing an architecture that can scale into dedicated processes will allow creating crowd simulations that are currently only possible in a non interactive mode. Our approach is based on separating the flow of control between three different threads, as described in Figure 4. Each of them is responsible for one specific component. For instance, we use a dedicated process for the 3D rendering, another one for the AI processing and a third one for the event handling and script executions. In order to fully take advantage of our multi-threaded approach, we need to keep the shared data as minimal as possible. Most current PC platforms like the Intel Pentium IV processor with Hyper-Threading or Dual-Core CPUs [Bin03] are capable of running two hardware threads simultaneously. By exploiting three active threads, we are optimizing the resource usage. One may argue that using one more active thread than the number supported by the processors may be suboptimal, but it occurs that the rendering thread is mainly bound on I/O operations, waiting for some acknowledgment from the GPU. The system architecture and performance also scope better with current trend in CPU hardware design, which features multiple simpler in-order cores ([Wan05], [Del]). By separating different components within our architecture, we are able to generate more complex simulations with the same level of interactivity.

3.3. Animation Blending

To obtain a more believable and realistic crowd, every agent in our system has a distinct behavior. To increase the variety, we are using a bank of animations simulating the different behaviors. Adding noise functions in the selection and the execution of every animation provides the feeling that every animation is unique. However, the animations transitions are clearly visible as the last and first frames of both animations do not match, introducing some jittering artifacts. To solve this problem, we have extended our animation system by introducing animation blending : rather than waiting for an animation completion, every agent may get a notification to blend their current animation with the next one. By varying the blending duration and by choosing different blending modes, we obtain more variety within the simulation. Figure 5 shows the animation transition over time.

4. Geometric LODs

To increase the number of displayed virtual humans, we render them using more or less simplified polygonal representations, *i.e.*, we choose the appropriate geometric LOD for each virtual human depending on its distance to the camera. The integration of a tool present in the OGRE3D [Ogr05] distribution into our asset pipeline allows us to easily create these LODs ([dHCSMT05]). However we had to adjust the tool to fit our needs. Indeed, we have observed some visual artifacts produced by the OGRE3D tool on the textures of our characters. We have fixed the tool by increasing the texture seam preservation, *i.e.*, by associating an infinite cost to the action of moving away from a seam. This fix allows us to reduce the geometry of every template until a limit of about 400 polygons. Going further than this limit introduces another kind of artifacts, this time on the produced geometry.

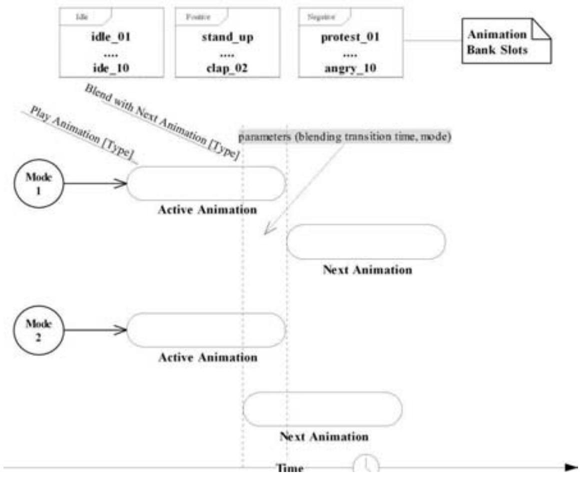


Figure 5: *Animation Blending Pipeline*

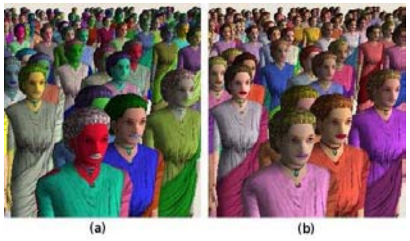


Figure 6: *Random color system (a) versus HSB control (b).*

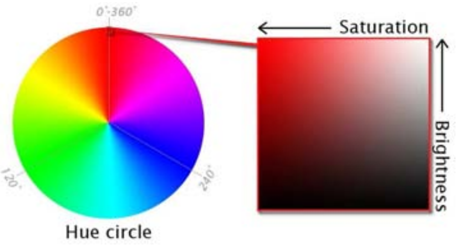


Figure 7: *HSB color space. Hue is represented by a circular region. A separate square region may be used to represent saturation and brightness, i.e., the vertical axis of the square indicates brightness, while the horizontal axis corresponds to saturation.*

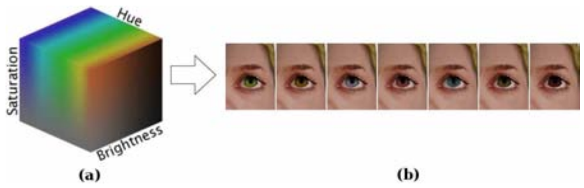


Figure 8: *The HSB space is constrained to a three dimensional color space with the following parameters (a): hue from 20 to 250, saturation from 30 to 80 and brightness from 40 to 100. Colors are then randomly chosen inside this space to add variety on the eyes texture of a character (b).*

5. Designing Variety

In the process of designing Romans and more generally human color variety, we must deal with localized constraints : some body parts need very specific colors. For instance, roman skin colors are taken from a specific range of unsaturated shades with red and yellow dominance, almost deprived of blue and green. Eyes are described as a range from brown to green and blue with different levels of brightness. These simple examples show that we cannot use a random color generator as is. We need a tool that allows us to control the randomness of color parameters for each body part of each roman (see Figure 6).

5.1. Color Models

The standard RGB color model representing additive color primaries of red, green, and blue is mainly used for specifying color on computer screens. In order to quantify and control the color parameters applied to the roman crowd, we need a user-friendly color model. Ray Smith ([Smi78]) proposes a model that deals with everyday life color concepts, *i.e.*, hue, saturation and brightness. This system is the HSB (or HSV) color model. The hue defines the specific shade of color, as a value between 0 and 360 degrees. The saturation denotes the purity of the color, *i.e.*, highly saturated colors are vivid while low saturated colors are washed-out like pastels. We represent saturation as a value between 0 and 100. The brightness measures how light or dark a color is, as a value represented between 0 and 100. The color space represented by the HSB model is shown in Figure 7.

5.2. HSB Color Model as a Tool for Designing Variety

We have found in the HSB color model an intuitive and flexible manner to control color variety. Indeed, as shown in Figure 8, by specifying a range for each of the 3 parameters, we are able to define a three-dimensional color space, that we call HSB map.

We have built a GUI so that a designer can easily load, modify and save HSB maps for different human templates (see Figure 10). This GUI provides all the necessary tools to efficiently :

- change the number of virtual humans rendered,
- select which virtual human template one wishes to work with,
- choose which texture of the selected template one wishes to work with,
- choose on which body part (alpha value) of the selected texture one wants to define color ranges,

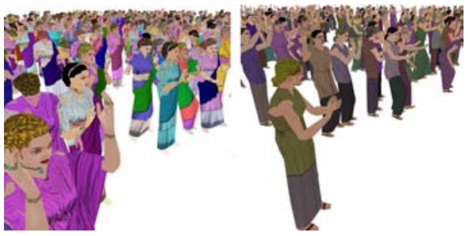
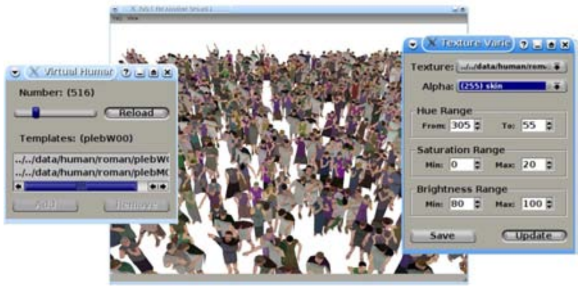


Figure 9: *Decurion women with saturation from 40 to 80 and brightness from 50 to 70 (left); plebeian women with saturation from 10 to 50 (right) .*



Number: (516)
Templates: (plebW00)
../../data/human/roman/plebW
../../data/human/roman/plebM
Texture: ../../data/human/rom
Alpha: (255) skin
Hue Range From: 305 To: 55
Saturation Range Min: 0 Max: 20
Brightness Range Min: 80 Max: 100

Figure 10: *Real-time design of lower classes texture variety. Dialog to select the template to edit and to choose the number of characters (left); the results displayed on the crowd (middle); dialog to design the variety of the selected template with the possibility to choose the body part along with the texture of the template to edit (right).*

- select a saturation and a brightness range between 0 and 100, and choose a range in the hue circle between 0 and 360 (cycles are allowed), for the currently selected alpha value.

The result of using these features is visualized in real time, *i.e.*, every change affects directly each rendered human.

5.3. Variety Case Study : Roman Society

To further illustrate the use of the GUI, we shortly present a case study in the framework of the ERATO Project [ERA05]. In order to simulate Roman society, we had to differentiate social classes. These differences were shown through clothing, where not only patterns, but colors as well were defining the rank of individuals. For instance, decurion women (rich elite) wore fine fabric with rich colors, while lower class citizens wore simple garments made of raw material usually dark. HSB maps allowed us to easily specify these significant differences by setting saturation and brightness values for rich garments and lower for modest ones (see Figure 9).

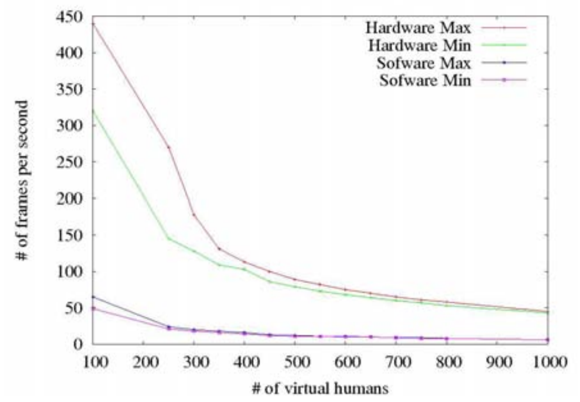


Figure 11: *Software and hardware rendering mode comparison (FPS on axis Y) using different number of visible characters (axis X).*

6. Results and discussion

For validating our multithreading design architecture, we have analysed our software using the Intel VTune Performance Environement [VTu05]. This toolkit allows to clearly investigate bottleneck issues for multi-threaded applications. The measurements were done on AMD64 4000+ with 2GB of memory and a nVidia SLI 6800 Ultra graphics card. The results depicts that our system is using an average of 75-80% of all the CPU cycles. According to [Cas04] those results are really interesting, especially considering that graphics applications are generally bound on I/O operations.

From Figure 11 we can see that the hardware shader approach of rendering the crowd completely outperforms the software solution even if we use an animation and geometry cache. In the tests we compared with one template consisting of 8 geometrical LODs, going from 1026 down to 490 triangles, playing two animations that were mixed at quaternion level using skeletons (33 bones) and consisting of 3 (512 pixels wide and high) textures. All humans were in view and the average of minimum and maximum frames per second were considered.

7. Conclusions and Future Work

By creating a virtual human character using our pipeline with LODs, texture varieties and color keying zones, animations and animation rules, this is multiplied into a diversified heterogeneous crowd, running fast on desktop computers. One template is thus re-usable for many different scenarios and settings with little or no changes to the data.

In the future we would like to extend the lighting model for different parts of the character on a per texel value using the fragment shader. Like this we can have special effects for the skin, clothes and jewelry for example. Moreover we will improve our tool for designing variety to handle high-resolution characters and work toward a custom geometric LOD creation tool.

We showed that shaders and multithreaded environment give a massive performance boost of around 700 percent consistently over the whole data set.

8. Acknowledgements

Special thanks to Mireille Clavien and Barbara Yersin for their help in designing and editing the materials present in this paper. The presented work is supported by the Swiss Federal Office for Education and Science and the EC FIFTH FRAMEWORK INCO-MED PROGRAMME, in frame of the EU ERATO project, number ICA3-CT-2002-10031.

References

[3DL05] 3dlabs, 2005. http://www.3dlabs.com.

[Bin03] BINSTOCK A.: Multithreading, hyper-threading, multiprocessing: Now, what's the difference? *IDS 1*, 1 (2003).

[Boo05] Boost C++ Library, 2005. http://www.boost.org.

[Cas04] CASEY S.: How to determine the effectiveness of hyper-threading technology with an application. *IDS 1*, 1 (2004).

[CLM05] COIC J.-M., LOSCOS C., MEYER A.: *Three LOD for the Realistic and Real-Time Rendering of Crowds with Dynamic Lighting*. Research Report RR-2005-008, Laboratoire d'InfoRmatique en Images et Systèmes d'information, Université Claude Bernard, France, 2005.

[Del] Powerpc 970m. http://www-306.ibm.com/chips/products/powerpc/.

[dHCSMT05] DE HERAS CIECHOMSKI P., SCHERTENLEIB S., MAÏM J., THALMANN D.: Reviving the roman odeon of aphrodisias: Dynamic animation and variety control of crowds in virtual heritage. In *Proc. 11th International Conference on Virtual Systems and Multimedia (VSMM 05)* (2005).

[dHCUDC04] DE HERAS CIECHOMSKI P., ULICNY B., D. T., CETRE R.: A case study of a virtual audience in a reconstruction of an ancient roman odeon in aphrodisias. In *Proc. 5th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST 04)* (2004).

[DHOO05] DOBBYN S., HAMILL J., O'CONOR K., O'SULLIVAN C.: Geopostors: A real-time geometry/impostor crowd rendering system. In *Proc. ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games* (2005).

[ERA05] ERATO - identification, evaluation and revival of the acoustical heritage of ancient theatres and odea, 2005. project website, http://www.at.oersted.dtu.dk// erato.

[GSM04] GOSSELIN D., SANDER P., MITCHELL J.: Rendering a crowd. In *ShaderX3 : Advanced Rendering with DirectX and OpenGL* (2004), Charles River Media, pp. 505–517.

[Kru05] KRUSZEWSKI P.: A practical system for real-time crowd simulation on current and next-generation gaming platforms.

[nVi05] nvidia, 2005. http://www.nvidia.com.

[Ogr05] Ogre3d, 2005. http://www.ogre3d.org.

[Ope05] OpenGL, 2005. http://www.opengl.org.

[Pho05] Adobe photoshop, 2005. http://www.adobe.com.

[RFD04] RYDER G., FLACK P., DAY A. M.: Adaptive crowd behaviour to aid real-time rendering of a cultural heritage environment. In *Proc. 5th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST 04)* (2004).

[Smi78] SMITH A. R.: Color gamut transform pairs. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1978), ACM Press, pp. 12–19.

[SWND03] SHREINER D., WOO M., NEIDER J., DAVIS T.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. Addison-Wesley, 2003.

[TCU*04] THALMANN D., CETRE R., ULICNY B., DE HERAS CIECHOMSKI P., CLAVIEN M.: Creating a virtual audience for the heritage of ancient theaters and odea. In *Proc. 10th International Conference on Virtual Systems and Multimedia (VSMM 04)* (2004).

[TLC02] TECCHIA F., LOSCOS C., CHRYSANTHOU Y.: Image-based crowd rendering. *IEEE Computer Graphics and Applications 22*, 2 (March-April 2002), 36–43.

[UdHCT04] ULICNY B., DE HERAS CIECHOMSKI P., THALMANN D.: Crowdbrush: interactive authoring of real-time crowd scenes. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation* (New York, NY, USA, 2004), ACM Press, pp. 243–252.

[VTu05] Vtune, 2005. http://www.intel.com.

[Wan05] WANG D.: The cell microprocessor. *ISSCC 1*, 1 (2005).

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1983), ACM Press, pp. 1–11.

[Wlo03] WLOKA M.: Batch, batch, batch: What does it really mean? *GDC 1*, 1 (2003).

[WS02] WAND M., STRASSER W.: Multi-resolution rendering of complex animated scenes. *Computer Graphics Forum 21*, 3 (2002). (Proc. Eurographics'02).