A fast and robust pipeline for populating mobile AR scenes with gamified virtual characters

Margarita Papaefthymiou[1,3], Andrew Feng[2], Ari Shapiro[2], George Papagiannakis[1,3]

[1]Foundation for Research and Technology Hellas, 100 N. Plastira Str., 70013, Heraklion, Greece
{mpapae02, papagian}@ics.forth.gr
[2]USC Institute for Creative Technologies, Playa Vista, CA, USA,
{feng, shapiro}@ict.usc.edu
[3]University of Crete, Computer Science Department, Voutes Campus,70013, Heraklion, Greece
papagian@csd.uoc.gr

Figure 1: *Mobile, AR, life-size gamified virtual characters powered through a fast, automatic animation pipeline with procedural body animation, speech and lip-sync.*

Abstract

In this work we present a complete methodology for robust authoring of AR virtual characters powered from a versatile character animation framework (Smartbody), using only mobile devices. We can author, fully augment with life-size, animated, geometrically accurately registered virtual characters into any open space in less than 1 minute with only modern smartphones or tablets and then automatically revive this augmentation for subsequent activations from the same spot, in under a few seconds. Also, we handle efficiently scene authoring rotations of the AR objects using Geometric Algebra rotors in order to extract higher quality visual results. Moreover, we have implemented a mobile version of the global illumination for real-time Precomputed Radiance Transfer algorithm for diffuse shadowed characters in real-time, using High Dynamic Range (HDR) environment maps integrated in our open-source OpenGL Geometric Application (glGA) framework. Effective character interaction plays fundamental role in attaining high level of believability and makes the AR application more attractive and immersive based on the SmartBody framework.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
SA'15 Symposium on MGIA, November 02 – 06, 2015, Kobe, Japan.
ACM 978-1-4503-3928-5/15/11.
http://dx.doi.org/10.1145/2818427.2818463

CR Categories: [Computer Graphics]: Graphics systems and interfaces - Mixed / augmented reality

Keywords: Augmented Reality, Rendering, Animation, Geometric Algebra, Illumination, mobile precomputed radiance transfer, procedural character animation systems

1 Introduction

Augmented Reality describes the technology that focuses on augmenting with computer virtual elements real environments using computer graphics and vision-based tracking and registration algorithms. Recently its popularity has increased due to the peace dividend from the mobile-devices competition wars.

In this work we present a novel pipeline for robust authoring in under 1 minute of Geometric and Photometric AR scenes consisting of animated virtual characters in a fast and robust manner. We use the glGA framework [Papagiannakis et al. 2014; Papagiannakis et al. 2015], a shader based C++ Computer Graphics (CG) framework integrated with a character animation platform, SmartBody [Shapiro 2011; Feng et al. 2014]. We have implemented the Precomputed Radiance Transfer (PRT) global illumination for real-time algorithm, running purely on mobile devices, for rendering diffuse unshadowed and shadowed virtual characters, integrated in our glGA framework. We present a marker-less AR authoring in indoors as well as in outdoors environments using only mobile

devices. Furthermore we complement our AR pipeline so that humanoid 3D models can be incorporated within seconds using the versatile SmartBody animation system that is infused with a wide range of capabilities, such as locomotion, object manipulation, gazing, speech synthesis and lip syncing. Finally we describe all the necessary steps so that any computer graphics framework can be easily integrated with SmartBody, in order to harness its advanced character behavioral capabilities.

In this work, for the first time, since all our previous virtual humans in AR [Papagiannakis et al. 2004; Egges et al. 2007] and in [Arnold et al. 2008; Jung and et al.] were based in offline created body and face animations that were played back in real-time, we have interactive virtual humans in AR via procedurally generated body and facial animation.

2 Previous Work

[Papagiannakis et al. 2004] successfully demonstrated a complete methodology for real-time mobile mixed reality systems with virtual character augmentations. The work featured realistic simulations of animated virtual human actors (clothes, body, skin, face) who augmented real environments (the archaeological site of ancient Pompeii) and re-enacted staged storytelling dramas. Although initially targeted at Cultural Heritage Sites, the paradigm was not limited to such subjects. However, portability, usability and form factor was a major impediment for wider adoption of that suite of technologies and algorithms.

Modern AR systems have been progressing since then [Papagiannakis et al. 2014; Arnold et al. 2008; Vacchetti et al. 2004; Egges et al. 2007; Gun and Billinghurst 2013] and already component-based frameworks have been researched for mobile outdoor applications [Papagiannakis et al. 2015; Huang et al. 2013].

[Langlotz et al. 2012] presents a novel system that allows in-place 3D content creation for mobile Augmented Reality in unprepared environments. The work described two different tracking techniques in order to create a feature database of the environment while the user runs the AR application. One tracking technique is for large working environments and the other for small workspaces. If the feature database exists, the user can retrieve it from the server with a query. [Gandy and MacIntyre 2014] presents the results of a user test of an AR toolkit, the DART system.

The SmartBody animation framework described in [Feng et al. 2014], provides a pipeline for incorporating high-quality humanoid assets into a virtual character and quickly infuse that character with a broad set of behaviors that are common to many games and simulations, including lip syncing [Xu et al. 2013] and nonverbal behavior [Marsella et al. 2013]. In this work we provide a novel integration of their framework for mobile AR scenes, featuring life-size, interactive virtual characters and a marker-less SLAM-based camera tracking system that allows the augmentation of any indoors or outdoors scene, in under one minute.

Finally, much previous work has been for Global Illumination using Precomputed Radiance Transfer Techniques (PRT) [Sloan et al. 2002; Nowrouzezahrai et al. 2011]. PRT is a real-time rendering method, which can be used for rendering diffuse and glossy not conformable objects in low frequency environments, with soft shadows and intereflections. For unshadowed diffuse illumination we used the method described in [Sloan et al. 2002]. This method is divided into two steps: a preprocess step and run time step. As a preprocess step functions are created over the object's surface that represent incident light into transferred radiance on each vertex of the object. At run time, these functions are applied to the incident light that comes from an environment map. Lighting and transfer functions are represented using low-order spherical harmonics.

3 Mobile AR Architecture based on glGA and SmartBody

The main framework used for our AR application is the open source OpenGL Geometric Application (glGA) framework. glGA is a lightweight, shader based C++ Computer Graphics (CG) framework which is developed for educational as well as research purposes. glGA is a cross platform application development framework and supports many mobile and desktop platforms. glGA contains many operations like compiling and loading shaders, textures, sounds, animations, loading 3D static meshes as well as rig meshes.

glGA supports from simple 3D models up to animated, skinned charcaters. In order to help the students visualize the externally rigged virtual characters (e.g. Collada or MD5 models) glGA provides the functionality required to parse the bone tree in real-time and retrieve the transformation matrix from each one of the joints. These matrices are then passed as uniform and vertex attribute parameters to the vertex shader. In the Table 1 and Figure 2 we provide the clear and concise overview of the glGA external dependencies. Seven well-known, well-documented and actively supported open-source libraries are utilized under the hood of glGA. The basic glGA application contains in a single file, only the main(), init() and display() C++ methods for maximum efficiency and readability.

Table 1: *glGA Open-source, external s/w libraries.*

GLFW	Window creation and OpenGL context handling
GLEW	OpenGL extension loading library
GLM	Template-based, C++ mathematics library similar to GLSL built-in math functions and types
AntTweakBar	GUI toolkit for widget creation and real-time shader/scene parameter handling based on user input
Image Magick	Multi-format Image/Texture loading
Assimp	Loading of static or skinned models
SmartBody	Character animation platform that provides locomotion, steering, object manipulation, lip syncing, gazing, physics, nonverbal behavior and other types of character movement in real time
GA Sandbox	Geometric Algebra operations
Free Image	open source library that supports many popular image formats like BMP, PNG, JPEG, TIFF as well as HDR and also provides tonemapping algorithms for HDR images.
Boost	set of libraries written in C++ that provides support for many functionalities like spherical harmonics, linear algebra, pseudorandom number generation, image processing, multithreading and regular expressions

We have developed glGA in such a way so that all of its examples and sample assignments can run in any of the standard desktop and mobile platforms: Windows, Linux, OSX and iOS. In order for all of those (10 in total) applications to be supported, we had to create a short Platform-Wrapper component that handles the platform specific functionality.

In addition to the desktop platforms of Windows, Linux and OSX, the glGA examples are also supported in the mobile iOS platform.

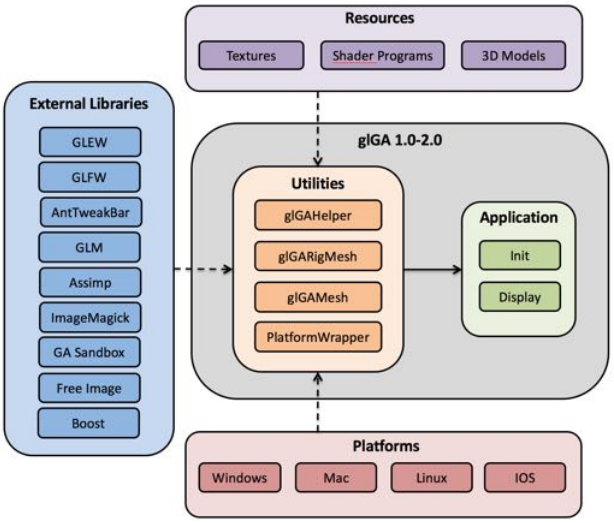

Figure 2: *The glGA overall framework s/w architecture.*

Here is where the PlatformWrapper is also employed not only due to the header files but also due to the different OpenGLES methods and calls (instead of standard OpenGL). An additional difference between desktop and mobile platforms is the way that external assets (e.g. textures, 3D models etc.) are loaded by the application. E.g. iOS uses bundles, while Windows, Linux and Mac retrieve the assets directly from the disk with either relative or absolute paths. Other than these, the current time retrieval is also different from desktop to mobile. E.g. it is essential during character animation, where we have to recalculate the matrix transformations based on the time passed since the animation started. As we have already, mentioned the code of the examples and assignments is in portable, standard C++, thus a standard C++ compiler (e.g. gcc, LLVM, Intel or Microsoft) is mandatory to be em-ployed. In glGA, we have also included some IDE project files for certain platforms already set up and ready to be built and executed. The project files that exist in glGA are for Visual Studio 2010 for Windows and Xcode 5.0.2 for Mac and iOS, while we also provide the basic gcc/g++ makefiles for Linux. The only modification required is to define the specific platform on top of the PlatformWrapper header file. Of course, other IDEs can also be used as long as they support standard C++. glGA can be accessed freely here: `http://george.papagiannakis.org/?page_id=513`.

Furthermore in glGA framework is integrated the Metaio SDK [Papagiannakis et al. 2014] framework. This framework is responsible for the AR functionalities and can perform markerless SLAM-based 3D camera tracking. The user can utilize the capabilities of the Metaio SDK in two ways. The user must create a 3Dmap using the Toolbox mobile application, and then pass this file to the application bundle using a Desktop File transfer application. Alternatively, he can create a 3D map at runtime. However, creating the 3D map with the second way has a limitation: you cannot extend the 3D map. In our work, we propose a way to load the 3D map created by Toolbox, directly from the device.

In our application the user must create the 3D map using the Toolbox mobile application, send it via e-mail, open the e-mail, select the file and open it via our application. In this way, we can have an extended 3D map directly in our application.

We achieve this by modifying the property list in the .plist file. The user must define the file type extension that the application will support (Uniform Type Identifier (UTI.)) by setting to the public.filename-extension3dmap. Also, we add the identifier for the custom UTI.When the user opens the .3dmap with the application the function: *-(BOOL) application:(UIApplication *) application openURL:(NSURL *) urlsource Application:(NSString *) source Application annotation:(id)annotation* in AppDelegate is called automatically. By using the variable url we extract the full path of the 3Dmap and set is as the metaioSDK configuration file.

The glGA framework in iOS platform initially supported rendering only for characters that consist of a single mesh, however, humanoids with clothes consist of multiple meshes. To achieve this we render each submesh of the character separately by creating a set of Vertex Attribute and Vertex Buffer Objects for each one of them. The desktop glGA used glDrawElementsBasevertex() function which is not supported in OpenGLES and we thus replaced it with glDrawArrays().

4 AR scene authoring with Geometric Algebra

We handle rotations of the AR scene objects with the use of Geometric Algebra (GA) by replacing euler angles with euclidean GA rotors. With the use of GA we produce more efficient results in terms of visual quality. GA rotors are simpler to manipulate than euler angles, more numerically stable and more efficient than rotation matrices. Moreover, GA rotors don't produce discontinuities to the rotation, by avoiding the problem of Gimbal Lock.

4.1 Review of the Euclidean Geometric Algebra model

GA [Dorst et al. 2010; Hestens and Sobczyk 1984] is a mathematical framework that provides a convenient mathematical notation for representing orientations and rotations of objects in three dimensions, a compact and geometrically intuitive formulation of algorithms, and an easy and immediate computation of rotors.

The basis vectors for the 3-dimensional Euclidean geometric algebra space are the orthonormal basis e_1, e_2 and e_3 which are the basis elements for generating the GA. The products of GA is the scalar product, the outer product, the inner product and the geometric product. The outer product, often called wedge product is denoted by $\wedge$ and has the properties of associativity, linearity and anti-symmetry, where a, b and c are GA vectors.

$Linearity : c \wedge (a + b) = (c \wedge a) + (c \wedge b)$
$Associativity : c \wedge (a \wedge b) = (c \wedge a) \wedge b$
$Anti - symmetry : a \wedge b = -b \wedge a$

The dot product is computed using the Equation 1.

$$a \wedge b = (a_1 e_1 + a_2 e_2 + a_3 e_3) \wedge +(b_1 e_1 + b_2 e_2 + b_3 e_3) \quad (1)$$

You can construct a higher level dimmensionality oriented subspace by defining the product between GA vectors. Such a subspace is called blade and a k-blade denotes a k-dimmensional subspace. For example, a vector is 1-blade, the outer product of 2 vectors is 2-blade, called bivector, the outer product of 3 vectors is 3-blade, called trivector etc. A bivector represents a plane and a trivector represents a 3D volume. The bivectors of the 3D euclidean GA are $e_1 \wedge e_2$, $e_2 \wedge e_3$, $e_3 \wedge e_1$ and the trivector $e_1 \wedge e_2 \wedge e_3$. The highest blade element is called pseudoscalar and is denoted by I. For example the pseudoscalar in 3D eucledian space is I_3.

The inner product, often called dot product is denoted by $\cdot$ and is used to compute distance and angles. The properties of the inner product is symmetry and linearity:

$Symmetry : a \cdot b = b \cdot a$
$Linearity : (ua + vb) \cdot c = u(a \cdot c) + v(b \cdot c)$

The inner product is computed using the Equation 2.

$$a \cdot b = |a||b|cos\phi \tag{2}$$

where ϕ
is the angle formed by the vectors a and b.

The geometric product between the vectors a and b (ab) equals to $a \cdot b + a \wedge b$. The geometric product is a mixed grade product: it consists of a scalar which is 0-blade and a bivector which is 2-blade, and is called multivector. The geometric product is computed using the Equation 3.

$$ab = a \cdot b + a \wedge b = |a||b|(cos\phi + Isin\phi = |a||b|e^{I\phi} \tag{3}$$

The duality of a GA element is denoted by $*$. The duality gives us a blade that represents the orthogonal complement of that subspase. For example, the duality of a bivector equals to the vector that is perpendicular to this bivector and vice versa. The duality is defined with the Equation 4.

$$A^* = A/I = -AI \tag{4}$$

For example the duality of the basis vector in the 3D space is computed as follows:

$e_2^* = e_2 I_3 = -e_2(e_1 \wedge e_2 \wedge e_3) = e_2 e_1 e_2 e_3 = -e_2 e_2 e_1 e_3 = -e_1 e_3 = -e_1 \wedge e_3$

The basic element used to handle rotations of any multivector in GA is rotor and is usually denoted as R. To rotate a multivector A we sandwiching it between the rotor R and its inverse rotor R^{-1} . As a result the rotated multivector is given by RAR^{-1}.

4.2 Algorithm description

Our main novelty lies in the replacement of euler angles with fast and robust GA rotors while the user rotates the objects of the AR scene.

As a first step we set the initial rotation of the scene on each axis and we compute the current GA rotor. We convert euler angles to quaternion representation and we compute angle and axis of the quaternion in order to compute the current GA rotor. To compute the rotor we use the following exponential 5.

$$R = e^{-I_3 u \frac{\phi}{2}} \tag{5}$$

where ϕ is the target angle of rotation and u is the axis of rotation. The code that computes the initial rotor of the AR scene is shown on Table 2.

Table 2: *Quaternion to GA rotor representation.*

```
quat destQ;
float angleDest = angle(destQ);
vec3 axisDest = axis(destQ);
mv v=unit_e(axisDest.x*e1+axisDest.y*e2+axisDest.z*e3);
rotor res =_rotor( exp( radians(angleDest)/2 * (-I3 * v)));
```

Our implementation gives the ability to the user to rotate the scene on global axis or on local axis. When we rotate on local axis we rotate the GA basis vectors with the current GA rotor in order to define the new local coordinate system. We rotate the basis vectors using the Equation ...The new rotated vectors rot_e_1, rot_e_2, rot_e_3 are used to define the new planes of rotation which are $rot\_e_1 \wedge rot\_e_2$, $rot\_e_2 \wedge rot\_e_3$ and $rot_e\_3 \wedge rot\_e_1$ on x, y, z axis respectively. In contrast, when we rotate in global axis the planes of rotation are $e_1 \wedge e_2$, $e_2 \wedge e_3$ and $e_3 \wedge e_1$ which extracted using the GA basis vectors. We use the Equation 6 to rotate the scene on an axis.

$$R = e^{-I_3 v} \tag{6}$$

where v is the plane of rotation.

5 Real time global illumination using PRT methods

One important part of the AR application is the global illumination, in order to produce more realistic results. The AR 3D objects must be illuminated base on the light of the real environment of the current scene that are located. To achieve this we use Precomputed Radiance Transfer (PRT) techniques. We integrated this functionality in the glGA framework. Our work so far includes diffuse unshadowed and shadowed PRT for static objects.

In our application the user has the ability to adjust the exposure in order to change the intensity of the light of the objects base on the light of the current scene. We give this functionality to the user because base on the time that is tracking, the light may have different intensity in comparison to the intensity captured in the environment map.

5.1 Shadowed-Transfer and Unshadowed-Transfer PRT implementation for mobile devices

The first step is to generate random samples using the Monte Carlo Integration. We divide the sphere's surface in RxR cells where $R = \sqrt{s}$ and s is the number of samples. To generate the samples we generate a random point in every cell. Then, we convert the points to vector coordinates and we compute the spherical harmonics coefficients for each sample. For the Spherical Harmonics implementation we used the Boost library.

The second step (precomputed) is to compute the coefficients that represent the incident radiance coming from the environment map for each vertex. We sum up for each sample, for each SH coefficient the multiplication of dot product of the vertex normal and the direction of the sample and the spherical harmonic value. The dot product gives positive value if the ray is inside the upper hemisphere. Then, we divide the result with the number of samples. For the shadowed- transfer PRT we multiply the dot product with the visibility term (1 if the vertex is visible otherwise 0). The Formula 7 is used for shadowed-transfer PRT, where ρ_x is the albedo at vertex x, $L_i(x, \omega_i)$ is the incoming light to vertex x from direction ω_i, $V(\omega_i)$ is the visibility term from direction ω_i and $N_x \cdot \omega_i$ is the dot product between vertex normal and direction ω_i.

$$L(x) = \frac{\rho_x}{\pi} \int_\Omega L_i(x, \omega_i) V(\omega_i) max(N_x \cdot \omega_i, 0) d\omega_i \tag{7}$$

We implement shadows by constructing Binary space partitioning (BSP) tree. BSP tree subdivide hierarchically a space into irregularly sized subspaces. Particularly, we use kd-trees. Constructing

and traversing the BSP trees and especially kd-trees is more efficient compared with other methods.

As a first step the kd-tree starts with a bounding box that contains all the primitives (triangles) of the scene. If the number of primitives in a bounding box (parent node) is greater than a predefined threshold then the particular bounding box is spitted into two different bounding boxes (child nodes). We split the bounding box along one of the coordinate axes (splitting is align with one of the axes). Triangles are associated with the bounding box in which are included in, and triangles that overlap two bounding boxes are associated with both of them. Each node of tree has a split axis and position and pointers to its two children.

Since we construct the kd-tree we are able to define if a vertex is illuminated or not from a certain direction by ray casting. We determine if the ray intersects any other triangle by traversing the kd-tree. We generate a ray form the vertex with direction towards the light. If the ray intersects any other triangle implies that the vertex is not illuminated from the particular direction.

At runtime, we compute the light coefficients base on the input environment map. Our environment maps are of HDR file format. We applied tonemapping to the environment map using the Free Image library. Firstly, we compute the light coming from the environment map for every generated sample by converting Cartesian coordinates of the sphere to image coordinates. Each light coefficient equals to the summation of the multiplication of light coming from the sample direction with the SH coefficient. We multiply the result with the probability $4 * PI$ and divide it with the number of samples.

The final color of each vertex equals to the summation of multiplication of the light coefficients with the vertex coefficients. The Fragment shader used for rendering PRT illumination is shown on Table 3.

Table 3: *Fragment shader for PRT illumunation.*

```
precision highp float ;
// texture of the character
uniform sampler2D texSampler;
// texture coordinate
varying vec2 fTexCoord;
// color of the vertex coming from environment map
varying vec3 fColor;
uniform float fExposure;
void main(){
      vec3 color = texture2D(texSampler , fTexCoord).rgb
                  * fColor * fExposure;
      gl_FragColor = vec4(color , 1.0);
}
```

5.2 Obtaining HDR Image Based Light Probes

The method that we use to obtain HDR light probes is capturing many HDR images from different viewing angles from a mirrored sphere. Then we crop the images using Photoshop (Elliptical Marquee Tool, select the area of the probe) and then loading the images in HDRshop. We convert the probe images to Latitude/Longitude format (Image-Panorama-Panoramic Transformations- Panoramic Transformations, destination image format: Latitude/Longitude) and we apply the analogue rotation (3D Rotation - Arbitrary Rotation - Settings) so that the images to be consistent. Then, we save the images in .hdr format, loading them in Photoshop and fix the errors of the most qualitative image using parts from the other angle images (i.e. we replace the part of the image that exists reflection by the tripod or the camera). Finally, we convert the new image back to mirrored ball format, using HDRshop. Our captured images and the generated final one is shown of Figure 3.

Figure 3: *The captured HDR mirror ball images (left) and the generated final one (right)*

Figure 4: *The initial character (right), unshadowed PRT (middle) and shadowed PRT (right) base the final generated probe on Figure 3.*

6 Integration of SmartBody to any modern shader-based CG framework

We integrated SmartBody with glGA in order to be able to have a wide range of behaviors in our characters and be able to use an animation designed for a certain character to another one (retargeting).

The main idea is to create a character in SmartBody with a deformable mesh instance, assign him/her an animation, take the transformation matrix on the current frame, of each bone of the character and pass it to the corresponding skeleton data structure in glGA. In this way, Smartbody can be integrated to any computer graphics framework in order to utilize their character animation generators.

We followed the following suggested SmartBody integration technique [Shapiro 2011; Feng et al. 2014]:

1. In your engine/simulation, create a character. Simultaneously, create a SmartBody character in the SmartBody context with the same name.
2. Implement the SBSceneListener interface which handles character creation/deletion and modification of SmartBody characters, and perform the equivalent actions in your engine/simulation.
3. Send SmartBody commands to control the character and change the scene every frame as needed.

4. Query SmartBody every frame to obtain the character state, and change the engine/simulation character's state to match it

Character behaviors such as speaking, gesturing, or other animated performance can be controlled by sending explicit commands using the Behavioral Markup Language (BML) [Kopp et al. 2006].

Firstly, we start the SimulationManager and we update it on every frame. We create a Character and a DeformableMesh (SmartBody classes) and we assign the second to the DeformableMeshInstance of the character. The next step is to create a Pawn, assign the skeleton of the character to that and assign the Pawn to the DeformableMeshInstance of the character. Also, we have to set the visibility of the DeformableMeshInstance of the character to true in order to Smartbody be able to update the transformation of the character. Then we run execBML function of the BMLProcessor in order to assign an animation to the character. On every frame we use the UpdateFast function of the DeformableMesh, we get the Joint names and Joint ids and we correlate them with the corresponding bones in glGA. Finally, we get the transformation matrix of each bone from the transform buffer of Smartbody and we assign them in the bones in glGA.

7 Results

In this section, we present our visual results of the AR applications in indoors as well as in outdoor environments.

On Figure 5 is shown the process of authoring the Geometric and Photometric AR scene in less than one minute. Firstly, we create a 3D map of the scene using Toolbox, and then we send it via e-mail and open it using our application. Next, we manage the size position and rotate of the character to achieve a life-sized augmentation and then we manage the shading of the object by adjust the exposure in order to be consistent with the scene. Finally, we save the transformations in order to have the same transformation when re-launching the app.

Figure 5: *Process of Geometric and Photometric AR scene authoring under one minute in outdoors (top) as well as indoors (bottom) environments.*

8 Conclusions and Future Work

In this work we proposed a method for robust authoring of Geometric and Photometric AR scenes in less than a minute. We handle rotations of the AR objects using GA rotors and we achieve higher quality resutls by avoiding the problem of Gimbal Lock. Moreover, we implemented a global illumination algorithm for unshadowed as well shadowed diffuse static objects using Precomputed Radiance Transfer methods, which we have integrated it in glGA framework. We have also shown the results of our AR app in indoors and in outdoor environments. Furthermore, we have integrated a character animation platform, SmartBody, with the glGA framework. Such integration will allow complex interactions with virtual characters through AR.

Our main focus is to expand and improve the robust, easy and fast AR authoring of 3D scenes involving both static as well as animated virtual characters, lit with natural scene HDR Image Based Light. In the future, we aim to give the ability to the user to interact with the virtual characters. Moreover, in the future, we could explore the other sensors (such as the heart rate, light, temperature sensors) in order to better connect the real-world with the virtual-world. We could also exploit data from the microphones, cameras or GPS. Lastly, an interesting thought would be creating a network of multiple secondary devices, providing and combining data from them.

Acknowledgements

The research leading to these results has received funding from the European Union People Programme (FP7-PEOPLE-2013-ITN) under grant agreement n^{O} 608013

References

ARNOLD, D., DAY, A., AND J. GLAUERT, E. A. 2008. Tools for populating cultural heritage environments with interactive virtual humans. *EPOCH Conference on Open Digital Cultural Heritage Systems*, 1–7.

DORST, L., FONTIJNE, D., AND MANN, S. 2010. *Geometric Algebra for Computer Science*. Morgan Kaufmann.

EGGES, A., PAPAGIANNAKIS, G., AND MAGNENAT-THALMANN, N. 2007. Presence and interaction in mixed reality environments. *Visual Computer 23*, 5, 317–333.

FENG, A., HUANG, Y., XU, Y., AND SHAPIRO, A. 2014. Fast, automatic character animation pipelines. *Computer Animation and Virtual Worlds 25*, 1 (Jan.), 3–16.

GANDY, M., AND MACINTYRE, B. 2014. Designers augmented reality toolkit, ten years later: Implications for new media authoring tools. *UIST*, 627–636.

GREEN, R. 2003. Spherical harmonic lighting: The gritty details. *Game Developers' Conference*.

GUN, A. L., AND BILLINGHURST, M. 2013. A component based framework for mobile outdoor ar applications. *In SIGGRAPH Asia 2013 Symposium on Mobile Graphics and Interactive Applications (SA '13)*, 173–179.

HESTENS, D., AND SOBCZYK, G. 1984. *Clifford Algebra to Geometric Calculus: A Unified Language for Mathematics and Physics*. Reidel Dordrecht.

HUANG, Z., HUI, P., PEYLO, C., AND D.CHATZOPOULOS. 2013. Mobile augmented reality survey: A bottom-up approach.

JUNG, Y., AND ET AL., A. K. D. F. Believable virtual characters in human-computer dialogs. *Eurographics 2011 - State of The Art Report*, 75–100.

KANBARA, M., AND YOKOYA, N. 2002. Geometric and photometric registration for real-time augmented reality. In *Proceedings of ISMAR2002*, 15–22.

KOPP, S., KRENN, B., MARSELLA, S., MARSHALL, A. N., PELACHAUD, C., PIRKER, H., THÓRISSON, K. R., AND VILHJÁLMSSON, H. 2006. Towards a common framework for multimodal generation: The behavior markup language. In *Intelligent virtual agents*, Springer, 205–217.

LANGLOTZ, T., MOOSLECHNER, S., ZOLLMANN, S., REITMAYR, C. D. G., AND SCHMALSTIEG, D. 2012. Sketching up the world: in situ authoring for mobile augmented reality. *Personal and Ubiquitous Computing 16*, 6, 623–630.

MARSELLA, S., XU, Y., LHOMMET, M., FENG, A., SCHERER, S., AND SHAPIRO, A. 2013. Virtual character performance from speech. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, 25–35.

NOWROUZEZAHRAI, D., GEIGER, S., MITCHELL, K., SUMNER, R., JAROSZ, W., AND GROSS, M. 2011. Light factorization for mixed-frequency shadows in augmented reality. *Computer Animation and Virtual Worlds*, 173–179.

PAPAGIANNAKIS, G., SCHERTENLEIB, S., PONDER, M., AREVALO, M., MAGNENAT-THALMANN, N., AND THALMANN, D. 2004. Real-time virtual humans in ar sites. *1st European Conference on Visual Media Production CVMP*, 273–276.

PAPAGIANNAKIS, G., PAPANIKOLAOU, P., GREASIDOU, E., AND TRAHANIAS, P. 2014. glga: an opengl geometric application framework for a modern, shader-based computer graphics curriculum. *Eurographics 2014*, 1–8.

PAPAGIANNAKIS, G., GREASIDOU, E., P.TRAHANIAS, AND TSIOUMAS, M. 2015. Mixed-reality geometric algebra animation methods for gamified intangible heritage. *International Journal of Heritage in the Digital 3*, 683–699.

SHAPIRO, A. 2011. Building a character animation system.

SLOAN, P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of ACM SIGGRAPH*.

SLOAN, P. 2008. Stupid spherical harmonics(sh) tricks. *GDC 2008*, 1–42.

VACCHETTI, L., LEPETIT, V., PONDER, M., PAPAGIANNAKIS, G., FUA, P., THALMANN, D., AND THALMANN, N. M. 2004. Stable real-time ar framework for training and planning in industrial environments. Virtual Reality and Augmented Reality Applications in Manufacturing, Ong, S. K., Nee, A.Y.C. (eds), ISBN: 1-85233-796-4, Springer-Verlag, 125–142.

XU, Y., FENG, A. W., MARSELLA, S., AND SHAPIRO, A. 2013. A practical and configurable lip sync method for games. In *Proceedings of Motion on Games*, ACM, 131–140.

APPENDIX A: Smart Body

SmartBody Invocation and calls

```
void createRetargetInstance(string srcSkelName,
                            string tgtSkelName){
    vector<string> endJoints=getEndJoints();
    vector<string> relativeJoints=getRelativeJoints();
    SBRetargetManager* retargetManager;
    retargetManager = m_pScene->getRetargetManager();
    SBRetarget* retarget;
    retarget = retargetManager->getRetarget(srcSkelName,
                                            tgtSkelName)
    retarget = retargetManager->createRetarget(srcSkelName,
                                               tgtSkelName);
    retarget->initRetarget(endJoints, relativeJoints);
}

void initializeSmartBody(){
    // set SB variables
    m_pScene = SBScene::getScene();
    assetManager = m_pScene->getAssetManager();
    sim = m_pScene->getSimulationManager();
    mediapath = PlatformWrapper::getMediapath();
    m_pScene->setMediaPath(mediapath);
    // load assets from mediapath
    m_pScene->loadAssetsFromPath(mediapath);

    //create jointmap for glGA character
    SBJointMap* RachelMap = new SBJointMap();
    SBJointMapManager* jointMapManager;
    jointMapManager = m_pScene->getJointMapManager();
    RachelMap = jointMapManager->createJointMap(mediapath
                                    + skeletonName);
    RachelMap->applySkeleton(m_pScene->getSkeleton(
                                        skeletonName));

    string RachelName = "Rachel";
    //apply glGA mesh and skeleton to SB character
    SBCharacter* Rachel;
    Rachel = m_pScene->createCharacter(RachelName,"Rachel");
    Rachel->setSkeleton(m_pScene->createSkeleton(skeletonName));
    Rachel->createStandardControllers();
    Rachel->setDoubleAttribute("deformableMeshScale",1);
    Rachel->setStringAttribute("deformableMesh","Rachel");
    string dMeshAttrib;
    dMeshAttrib = Rachel->getStringAttribute("deformableMesh");
    Rachel->setStringAttribute("deformableMesh","dMeshAttrib");
    Rachel->dMeshInstance_p = new DeformableMeshInstance();
    Rachel->dMeshInstance_p->setDeformableMesh(
                assetManager->getDeformableMesh(meshName));
    Rachel->dMeshInstance_p->setPawn(Rachel);
    //mapping glGA character joints - motion
    auto_map(RachelMap);
    SBJointMap *zebra2Map=m_pScene->getJointMapManager()
                          ->createJointMap("zebra2");
    //mapping SB character joints - motion
    zebra2_map(zebra2Map);
    //create SB skeleton
    SBSkeleton* bradSkeleton=m_pScene->skeletonSB(skeletonSB);
    zebra2Map->applySkeleton(bradSkeleton);
    SBMotion *motion = m_pScene->getMotion("ChrMarine@Walk01");
    motion->setMotionSkeletonName(skeletonSB);
    zebra2Map->applyMotion(motion);
    SBSkeleton skelName = m_pScene->getCharacter(RachelName)
                        ->getSkeleton();
    //retarget a motion that is created
    //for SB skeleton to glGA skeleton
    createRetargetInstance(skeletonSB,skelName.getName());
    //start simulation manager
    sim->start();
    sim->setupTimer();
    //Rachel executes ChrMarine@Walk01 animation
    m_pScene->getBmlProcessor()->execBML("Rachel",
            "<body posture=\"ChrMarine@Walk01\"/>");
}

void updateSB(){
   // update SB animations
   m_pScene->update();
   sim->updateTimer();
```

```
    updateJoints();
}
```

SmartBody to glGA conversion for every frame

```
void updateJoints(){
    // update joints of SB character
    char->dMeshInstance_p->updateTransformBuffer();
    vector<string> names=char->dMeshInstance_p
                            ->getJointNames();
    map<string,int> ids=char->dMeshInstance_p
                            ->getJointIds();

    // foreach joint
    for (int i=0; i<names.size(); i++){
        if(boneMapping.find(names[i]) != boneMapping.end()){
            // index of joint to glGA
            int index = boneMapping[names[i]];
            // get transformation matrix of the joint
            SrMat sb_mat_tmp=char->dMeshInstance_p
                  ->transformBuffer.at(ids[names[i]]);
            mat4 mat4tmp = mat4(
            sb_mat_tmp.e11(),sb_mat_tmp.e12(),
            sb_mat_tmp.e13(),sb_mat_tmp.e14(),
            sb_mat_tmp.e21(),sb_mat_tmp.e22(),
            sb_mat_tmp.e23(),sb_mat_tmp.e24(),
            sb_mat_tmp.e31(),sb_mat_tmp.e32(),
            sb_mat_tmp.e33(),sb_mat_tmp.e34(),
            sb_mat_tmp.e41(),sb_mat_tmp.e42(),
            sb_mat_tmp.e43(),sb_mat_tmp.e44());
            boneInfo[index].finalTransformation = mat4tmp;
         }
    }
}
```

APPENDIX B: PRT

Random Sampling - set SH coefficients

```
int samples, bands;
vector<SHSample> samplesCoefficients;
int numCoeff=bands*bands;
vector<SHSample> setCoefficients(){
    int sqrtSamples = std::sqrt(samples);
    for (int i=0; i<sqrtSamples; i++){
        for (int j=0; j<sqrtSamples; j++)
        {
            // generate random samples on the sphere
            double x=(i+(double)rand()/RAND_MAX)/sqrtSamples;
            double y = (j+(double)rand()/RAND_MAX)/sqrtSamples;
            double theta = 2.0f*acos(sqrt(1.0f-x));
            double phi=2.0f*M_PI*y;
            SHSample cur;
            cur.sphereCoord=vector3(theta,phi,1.0) ;
            cur.vectorCoord=vector3(sin(theta)*cos(phi),
                                    sin(theta)*sin(phi),
                                    cos(theta));
            cur.coeff = new double[bands*bands];
            //set SH coefficients for every sample
            for(int l=0; l<bands; l++){
                for(int m=-l; m<=l; m++){
                    int index=l*(l+1)+m;
                    complex<double> SH;
                    SH= spherical_harmonic(l, m, theta, phi);
                    cur.coeff[index]=SH.real();
                }
            }
            samplesCoefficients.push_back(cur);
        }
    }
    return samplesCoefficients;
}
```

Compute Light Coefficients

```
vec3* setLightCoeffs(){
    vec3* lightCoeffs;
    lightCoeffs=new vec3[numCoeff];
    for(int i = 0; i < numCoeff; i++){
        lightCoeffs[i]=vec3(0.0,0.0,0.0);
        for(int j=0; j < samples; j++){
            //incoming color on enviroment map from sample dir
            vec3 light=this->getLight(samplesCoefficients[j]);
            double curC=samplesCoefficients.at(j).coeff[i];
            lightCoeffs[i]+=vec3(light[0] * curC,
                        light[1] * curC,light[2]* curC);
        }
        //devide by num of samples
        lightCoeffs[i]*=(4.0f*M_PI/samples);
    }
    return lightCoeffs;
}
```

Compute Vertex Coefficients

```
void vertCoefficients(){
    vertCoeff = new double[perVertData.size() * numCoeff];
    int numSamples = (samplesCoefficients).size();
     // for each vertex
     for (unsigned int i = 0; i < perVertData.size(); i++){
         double curVertCoeff[numCoeff];
            // for each sample
            for (int j=0; j< numSamples; j++){
              // dot product of normal and sample direction
              //kdtree to compute visibility of vertex
              double brightness;
              brightness = computeBrightness(
                          perVertData[i].normal
                          perVertData[i].position,
                          samplesCoefficients[j],kdTree);
              if (brightness > 0.0)
                 //add brightness*SH foreach sample,
                 //for each coefficient
                 for(int m=0; m<numCoeff; m++){
                    double val = (brightness*
                          samplesCoefficients.at(j).coeff[m]);
                    curVertCoeff[m]+=val;}
            }
            double factor = 4.0f*M_PI / numSamples;
            for(int j=0; j<numCoeff; j++){
                curVertCoeff[j] = curVertCoeff[j] * factor;
                vertCoeff[i*numCoeff+j] = curVertCoeff[j];
            }
     }
     //write to file coefficients for each vertex
     writeToFile();
}
```

Compute color for each vertex

```
void computeColors(){
    //foreach vertex
    for (int i=0; i<data.size(); i++){
        vec3 curColor(0.0,0.0,0.0);
        for (int j=0; j<numCoeff; j++){
            vec3 b = lightCoeffs[j];
            double a = vertCoeff[i*numCoeff + j];
            //color=light coeff * vertex coeff
            vec3 color = vec3(b[0]*a,b[1]*a, b[2]*a);
            // add colors foreach coefficient
            curColor +=  color;
        }
        vec3 finalColor(curColor[0]/M_PI,
             curColor[1]/M_PI,curColor[2]/M_PI);
        data[i].color = finalColor;
    }}
```