Dynamically Creating Virtual Museums

DIONYSIOS POLITIS, GERASIMOS VAIOU, MARRAS IOANNIS, PAPAIOANNOU ATHANASIOS, LAFTSIDIS CHARALAMPOS

Department of Informatics

Aristotle University of Thessaloniki
University Campus, Thessaloniki, GR-541 24
GREECE

Abstract: - The idea of creating a virtual museum is far from new. However, creating a museum that an archaeologist could customize to match his needs is quite innovative.

Here we present a system which can be used for online visualization of museums. Although, there are plenty of online virtual museums, none of them is customizable. These museums are designed statically and represent certain museums which makes it rather difficult to change.

On the contrary, our Dynamic Virtual Museum is easily managed through database entries, which provide all necessary variables (rooms, models, exhibits) and interact with the renderer through scripts. Therefore, the virtual museum can be easily transformed to match any given exhibition or a visitor's specific choices.

Key-Words: - Virtual Museum, Dynamic, Web, Databases.

1 Introduction

The system consists of two main elements, a database where all information about the exhibits, models etc. is kept and a renderer which is responsible for graphically representing all this information on the computer screen.

The database part is handled by MySQL[1], whereas VRML[4] is responsible for all the graphics. In order to easily connect MySQL and VRML through an easy to use web interface, php[3] is used.

Our goal is to create a fully customizable museum, which will be easy to navigate and control by any archaeologist or visitor. It could be used in a wide range of occasions, such as an exhibition centre where exhibits are changed quite often, it could be an important help to a museum executive who needs to rearrange some or all of the exhibits, or it could be used by any visitor who wants an exhibit to match his certain needs.

2 The renderer

The graphics subsystem of the application is responsible for rendering the museum rooms and the exhibits displayed therein. It also handles the interaction of the user with the virtual world. This would allow, for example, a verbal description to be played back,

Figure 1: A sculpture and some paintings in a virtual museum

whenever the user clicks on an exhibit.

The render is implemented using VRML (Virtual Reality Modeling Language), a web-based network protocol for working with three dimensional (3D) scenes or data sets.

It allows the creation ofplatform-independent 3D objects, described in text files, which can then be displayed on any computer platform for which an appropriate browser exists.

VRML browsers come in two types: stand-alone and plugins for HTML browsers.They allow a user to walk into a VRML scene using a mouse or keyboard and navigate, as he does in the real world. A VRML document, like an html document, is a formalized text description of a Web page's contents. Unlike html however, VRML is not "marked up" text. It contains descriptions of three-dimensional objects and their interrelationships.

3 Database

The database subsystem of the application is responsible for the storage of all elements important to the museum model. It is organized in a way which makes it possible for the system to easily extract information about the exhibits and how they should be displayed in the virtual museum.

The database used is MySQL, the most commonly used database in conjunction with php. It's architecture makes it extremely fast and easy to customize.

Using a database it is quite a simple task to record a large number of data and information about an object, without necessarily having to use them all in the construction of the model or the display of the object in the museum. Therefore, we can create a well organized library of all of our artifacts and exhibits.

The most important records that describe an object, should be the object's type (whether it's a painting, a sculpture, a mask etc.), its measurements (height, width, weight etc.), a title (if there is one ex. "Mona

Lisa") and a short description of the artifact. The description could be simple text or even a path (relative or not) to an audio file, which could be used in the museum model. The visitor then could hear a narrated description of the artifact, by interacting with it.

Other characteristic features that could be recorded as well are the artifacts' distinguishing features, creation date or period (if known), its origin, maker's name and the materials which were used. These descriptive items could be easily added even after the creation of the database through the web interface or any other administration tool for MySQL databases. Such a tool is *eskuel*[1], a MySQL databases administration interface written in PHP. It allow user to manage simply and fully one or more database without any advanced knowledge in SQL language.

4 Scripting

VRML was created for describing interactive, but static, 3D objects and worlds. Therefore, there was no need for variables when the specification was written. When creating a dynamic virtual museum you need to be able to process data and change many of the models' attributes (size, translations, geometry, materials etc.). You must also have the capability of extracting specific fields from a database record and provide the field values to the VRML model. Hence, the need for a scripting language to solve these problems was born. For all the scripting tasks, php is used.

PHP is an HTML-embedded scripting language. Much of its syntax is borrowed from C, Java and Perl with a couple of unique PHP-specific features thrown in. It has built-in functions that allow you to perform various functions on a MySQL database and can be used to solve complex mathematical equations using libbcmath which is bundled with PHP (since version 4.0.4). Both of these characteristics made the use of php for the virtual museum an easy choice.

5 Mathematics

Coordinate {
exposedField MFVec3f point [ ] (-INF,INF)
}

This node defines a set of 3D coordinates to be used in the coord field of vertex-based geometry nodes including IndexedFaceSet, IndexedLineSet, and PointSet.

The VRML 2.0 naming philosophy is to give each node the most obvious name and not try to predict how the specification will change in the future. If carried out to its logical extreme, then a philosophy of planning for future extensions might give Coordinate the name CartesianCoordinate3Float, since support for polar or spherical coordinates might possibly be added in the future, as might double-precision or integer coordinates.

ElevationGrid

```
ElevationGrid {
eventIn      MFFloat set_height
exposedField SFNode  color           NULL
exposedField SFNode  normal          NULL
exposedField SFNode  texCoord        NULL
field        MFFloat height          []
field        SFBool  ccw             TRUE
field        SFBool  colorPerVertex  TRUE
field        SFFloat creaseAngle     0
field        SFBool  normalPerVertex TRUE
field        SFBool  solid           TRUE
field        SFInt32 xDimension      0
field        SFFloat xSpacing        1.0
field        SFInt32 zDimension      0
field        SFFloat zSpacing        1.0
```

The ElevationGrid node specifies a uniform rectangular grid of varying height in the Y=0 plane of the local coordinate system.

[1] http://www.phptools4u.com/scripts/eskuel/?lang=english

The geometry is described by a scalar array of height values that specify the height of a surface above each point of the grid.

The xDimension and zDimension fields indicate the number of elements of the grid height array in the X and Z directions. Both xDimension and zDimension must be greater than or equal to zero. The vertex locations for the rectangles are defined by the height field and the xSpacing and zSpacing fields:

Thus, the vertex corresponding to the point P[i, j] on the grid is placed at:

P[i,j].x = xSpacing / i
P[i,j].y = height[i + j / xDimension]
P[i,j].z = zSpacing / j

where $0 <= i < xDimension$ and $0 <= j < zDimension$, and $P[0, 0]$ is $height[0]$ units above/below the origin of the local coordinate system.

The colorPerVertex field determines whether colours specified in the colour field are applied to each vertex or each quadrilateral of the ElevationGrid node. If colorPerVertex is FALSE and the color field is not NULL, the color field shall specify a Color node containing at least $(xDimension - 1)/(zDimension - 1)$ colours; one for each quadrilateral, ordered as follows:

$QuadColor[i, j] = Color[i+j/(xDimension-1)]$

where $0 <= i < xDimension - 1$ and $0 <= j < zDimension - 1$,
and $QuadColor[i, j]$ is the colour for the quadrilateral
defined by $height[i + j/xDimension]$,
$height[(i + 1) + j/xDimension]$,
$height[(i + 1) + (j + 1)/xDimension]$ and
$height[i + (j + 1)/xDimension]$

If colorPerVertex is TRUE and the color field is not NULL, the color field shall specify a Color node containing at least xDimension / zDimension colours, one for each vertex, ordered as follows:

$VertexColor[i, j] = Color[i+j/xDimension]$

where $0 <= i < xDimension$ and $0 <= j < zDimension$, and $VertexColor[i, j]$ is the colour for the vertex defined by $height[i+j/xDimension]$

The normalPerVertex field determines whether normals are applied to each vertex or each quadrilateral of the ElevationGrid node depending on the value of normalPerVertex. If normalPerVertex is FALSE and the normal node is not NULL, the normal field shall specify a Normal node containing at least $(xDimension-1)/(zDimension-1)$ normals; one for each quadrilateral, ordered as follows:

$QuadNormal[i, j] = Normal[i + j/(xDimension - 1)]$

where $0 <= i < xDimension - 1$ and $0 <= j < zDimension - 1$, and $QuadNormal[i, j]$ is the normal for the quadrilateral defined by $height[i + j/xDimension]$, $height[(i + 1) + j/xDimension]$, $height[(i + 1) + (j + 1)/xDimension]$ and $height[i + (j + 1)/xDimension]$

If normalPerVertex is TRUE and the normal field is not NULL, the normal field shall specify a Normal node containing at least xDimension / zDimension normals; one for each vertex, ordered as follows:

$VertexNormal[i, j] = Normal[i + j/xDimension]$

where $0 <= i < xDimension$ and $0 <= j < zDimension$, and $VertexNormal[i, j]$ is the normal for the vertex defined by $height[i + j/xDimension]$

The texCoord field specifies per-vertex

texture coordinates for the ElevationGrid node. If texCoord is NULL, default texture coordinates are applied to the geometry. The default texture coordinates range from (0,0) at the first vertex to (1,1) at the last vertex. The S texture coordinate is aligned with the positive X-axis, and the T texture coordinate with positive Z-axis. If texCoord is not NULL, it shall specify a TextureCoordinate node containing at least $(xDimension)/(zDimension)$ texture coordinates; one for each vertex, ordered as follows:

$VertexTexCoord[i, j] =$
$TextureCoordinate[i + j/xDimension]$

where $0 <= i < xDimension$ and $0 <= j < zDimension$, and $VertexTexCoord[i, j]$ is the texture coordinate for the vertex defined by $height[i + j/xDimension]$

Group

```
Group {
eventIn      MFNode  addChildren
eventIn      MFNode  removeChildren
exposedField MFNode  children   []
field        SFVec3f bboxCenter 0 0 0
field        SFVec3f bboxSize  -1 -1 -1
}
```

A Group node contains children nodes without introducing a new transformation. It is equivalent to a Transform node without the transformation fields.

The bboxCenter and bboxSize fields specify a bounding box that encloses the Group node's children. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, the results are undefined. A default bboxSize value, (-1, -1, -1), implies that the bounding box is not specified and, if needed, is calculated by the browser.

IndexedFaceSet

```
IndexedFaceSet {
eventIn       MFInt32 set_colorIndex
eventIn       MFInt32 set_coordIndex
eventIn       MFInt32 set_normalIndex
eventIn       MFInt32 set_texCoordIndex
exposedField  SFNode  color           NULL
exposedField  SFNode  coord           NULL
exposedField  SFNode  normal          NULL
exposedField  SFNode  texCoord        NULL
field         SFBool  ccw             TRUE
field         MFInt32 colorIndex      []
field         SFBool  colorPerVertex  TRUE
field         SFBool  convex          TRUE
field         MFInt32 coordIndex      []
field         SFFloat creaseAngle     0
field         MFInt32 normalIndex     []
field         SFBool  normalPerVertex TRUE
field         SFBool  solid           TRUE
field         MFInt32 texCoordIndex   []
}
```

The IndexedFaceSet node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the coord field. The coord field contains a Coordinate node that defines the 3D vertices referenced by the coordIndex field. IndexedFaceSet uses the indices in its coordIndex field to specify the polygonal faces by indexing into the coordinates in the Coordinate node. An index of "-1" indicates that the current face has ended and the next one begins. The last face may be (but does not have to be) followed by a "-1" index. If the greatest index in the coordIndex field is N, the Coordinate node shall contain N+1 coordinates (indexed as 0 to N). Each face of the IndexedFaceSet shall have:

1. at least three non-coincident vertices,
2. vertices that define a planar polygon,
3. vertices that define a non-self-intersecting polygon.

Otherwise, results are undefined.

IndexedLineSet

```
IndexedLineSet {
eventIn       MFInt32 set_colorIndex
eventIn       MFInt32 set_coordIndex
exposedField  SFNode  color           NULL
```

```
exposedField  SFNode  coord          NULL
field         MFInt32 colorIndex     []
field         SFBool  colorPerVertex TRUE
field         MFInt32 coordIndex     []
}
```

The IndexedLineSet node represents a 3D geometry formed by constructing polylines from 3D vertices specified in the coord field. IndexedLineSet uses the indices in its coordIndex field to specify the polylines by connecting vertices from the coord field. An index of "-1" indicates that the current polyline has ended and the next one begins. The last polyline may be (but does not have to be) followed by a "-1". IndexedLineSet is specified in the local coordinate system and is affected by ancestors' transformations.

OrientationInterpolator

```
OrientationInterpolator {
eventIn      SFFloat    set_fraction
exposedField MFFloat    key           []
exposedField MFRotation keyValue      []
eventOut     SFRotation value_changed
}
```

The OrientationInterpolator node interpolates among a set of rotation values specified in the keyValue field. These rotations are absolute in object space and therefore are not cumulative. The keyValue field shall contain exactly as many rotations as there are keyframes in the key field.

An orientation represents the final position of an object after a rotation has been applied. An OrientationInterpolator interpolates between two orientations by computing the shortest path on the unit sphere between the two orientations. The interpolation is linear in arc length along this path. If the two orientations are diagonally opposite results are undefined.

If two consecutive keyValue values exist such that the arc length between them is greater than PI, the interpolation will take place on the arc complement. For example, the interpolation between the orientations $(0, 1, 0, 0)$ and $(0, 1, 0, 5.0)$ is equivalent to the rotation between the orientations $(0, 1, 0, 2\pi)$ and $(0, 1, 0, 5.0)$.

Shape

Shape
exposedField SFNode appearance NULL
exposedField SFNode geometry NULL

The Shape node has two fields, appearance and geometry, which are used to create rendered objects in the world. The appearance field contains an Appearance node that specifies the visual attributes (e.g., material and texture) to be applied to the geometry. The geometry field contains a geometry node. The specified geometry node is rendered with the specified appearance nodes applied.

If the geometry field is NULL, the object is not drawn.

TextureCoordinate

TextureCoordinate
exposedField MFVec2f point []

The TextureCoordinate node specifies a set of 2D texture coordinates used by vertex-based geometry nodes (e.g., IndexedFaceSet and ElevationGrid) to map textures to vertices. Textures are two dimensional colour functions that, given an (s, t) coordinate, return a colour value colour(s, t). Texture map values (ImageTexture, MovieTexture, and PixelTexture) range from $[0.0, 1.0]$ along the S-axis and T-axis. However, TextureCoordinate values, specified by the point field, may be in the range $(-\infty, \infty)$. Texture coordinates identify a location (and thus a colour value) in the texture map. The horizontal coordinate s is specified first, followed by the vertical coordinate t.

If the texture map is repeated in a given direction (S-axis or T-axis), a texture coordinate C (s or t) is mapped into a texture map that has N pixels in the given direction as follows:

$Texturemaplocation =$
$= (C - floor(C))/N$

If the texture map is not repeated, the texture coordinates are clamped to the 0.0 to 1.0 range as follows:

$Texturemaplocation =$
$= N, if C > 1.0,$
$= 0.0, if C < 0.0,$
$= C/N, if 0.0 <= C <= 1.0.$

TextureTransform

```
TextureTransform {
exposedField SFVec2f center      0 0
exposedField SFFloat rotation    0
exposedField SFVec2f scale       1 1
exposedField SFVec2f translation 0 0
}
```

The TextureTransform node defines a 2D transformation that is applied to texture coordinates (see 3.48 TextureCoordinate). This node affects the way textures coordinates are applied to the geometric surface. The transformation consists of (in order):
1.a translation,
2.a rotation about the centre point,
3.a non-uniform scale about the centre point.

In matrix transformation notation, where Tc is the untransformed texture coordinate, Tc' is the transformed texture coordinate, C (center), T (translation), R (rotation), and S (scale) are the intermediate transformation matrices,
$Tc' = -C/S/R/C/T/Tc$

Transform

```
Transform {
eventIn      MFNode      addChildren
eventIn      MFNode      removeChildren
exposedField SFVec3f     center           0 0 0
exposedField MFNode      children         []
exposedField SFRotation  rotation         0 0 1 0
exposedField SFVec3f     scale            1 1 1
exposedField SFRotation  scaleOrientation 0 0 1 0
exposedField SFVec3f     translation      0 0 0
field        SFVec3f     bboxCenter       0 0 0
field        SFVec3f     bboxSize         -1 -1 -1
}
```

The Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors.

The bboxCenter and bboxSize fields specify a bounding box that encloses the children of the Transform node. This is a hint that may be used for optimization purposes. If the specified bounding box is smaller than the actual bounding box of the children at any time, the results are undefined. A default bboxSize value, (-1, -1, -1), implies that the bounding box is not specified and, if needed, must be calculated by the browser. A description of the bboxCenter and bboxSize fields is provided in "2.6.4 Bounding boxes."

The translation, rotation, scale, scaleOrientation and center fields define a geometric 3D transformation consisting of (in order):
1. a (possibly) non-uniform scale about an arbitrary point
2. a rotation about an arbitrary point and axis
3. a translation

The center field specifies a translation offset from the origin of the local coordinate system $(0, 0, 0)$. The rotation field specifies a rotation of the coordinate system. The scale field specifies a non-uniform scale of the coordinate system. scale values shall be > 0.0. The scaleOrientation specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The scaleOrientation applies only to the scale

operation. The translation field specifies a translation to the coordinate system.

The translation/rotation/scale operations performed by the Transform node occur in the "natural" order each operation is independent of the other.

Given a 3-dimensional point P and Transform node, P is transformed into point P' in its parent's coordinate system by a series of intermediate transformations. In matrix transformation notation, where C (center), SR (scaleOrientation), T (translation), R (rotation), and S (scale) are the equivalent transformation matrices,
$P' = T/C/R/SR/S/-SR/-C/P$

The second operation, in order, is the scaleOrientation.The scaleOrientation temporarily rotates the object's coordinate system (i.e., local origin) in preparation for the third operation, scale, and rotates back after the scale is performed.

The fourth operation is rotation. It specifies an axis about which to rotate the object and the angle (in radians) to rotate.

The last operation is translation. It specifies a translation to be applied to the object.

"Spaces" is kindof like a coordinate system. Well, it actually *could* be a coordinate system, but not always, though in your case it most likely will be. Also, the system I am describing follows the "Camera moves in space" concept, sure the math is all the same no matter what you do, but *how* the math is implemented can lead to interesting problems if you accidentily switch between "camera moves" and "camera stays still".
Definitions:
Object Space:
The vertex data of an object usually set up so that $0, 0, 0$ is the center of rotation you desire. World Space:
A space where all object's vertex data(for all objects) are represented with respect to ONE coordinate system centered at $0, 0, 0$. View Space:
Almost like a world space, but what happens is you translate the camera to the World Space 0,0,0 position and align it to the cardinal axes. All of the objects in the world are moved such that thay are in their relative locations and orientations with respect to the camera(which is now at $0, 0, 0$). Screen Space:
This is the result you get when you project the View Space Coordinates of the vertex data for each object via projection equations.
Rotations:

```
Rotation about the X axis by an angle a:
|1       0        0    0|
|0  cos(a)  -sin(a)    0|
|0  sin(a)   cos(a)    0|
|0       0        0    1|

Rotation about the Y axis by an angle a:
| cos(a)  0  sin(a)    0|
|      0  1       0    0|
|-sin(a)  0  cos(a)    0|
|      0  0       0    1|

Rotation about the Z axis by an angle a:
|cos(a)  -sin(a)  0   0|
|sin(a)   cos(a)  0   0|
|     0        0  1   0|
|     0        0  0   1|

Projection Matrix:
|1   0   0   0|   |
|0  -1   0   0| * |
|0   0 1/d   0|   |
|0   0   0   1|   |

           |   |x|   | x' |
 transform | * |y| = |-y' |
 matrix    |   |z|   |z'/d|
           |   |1|   |  1 |

d =W / 2*tan(a/2)
```

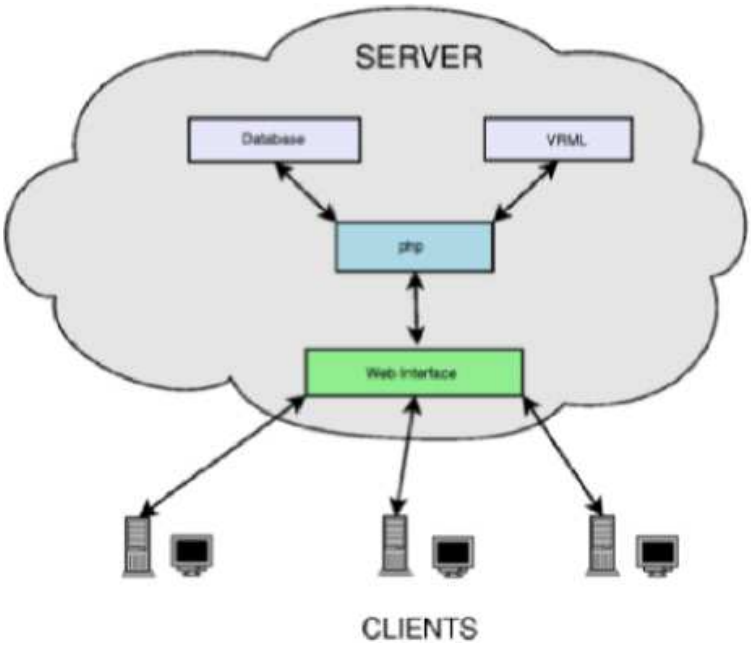

Figure 2: VRML, MySQL and php connection diagram

W = screen width in pixels
a = a desired Field of View (normally $\pi/3$ to π rad)

6 Web interface

All the tasks concerning the administrating or use of the virtual museum, are completed through an easy to use web interface. Firstly, one should login either as an administrator or a simple user to identify himself to the system.

After an administrator is identified several options are presented. An administrator can add or remove new artifacts in the database. He could also view and/or modify an existing record. Viewing all the records should be considered an obvious option, given the built-in MySQL functions that php has, thus making it possible for the administrator to sort and view all record by their id number, type, title or any other field. An administrator can also, after viewing the first, automatically constructed model, rearrange manually the position of certain artifacts.

A visitor can only view the existing records in the database. Even then, specific fields could be hidden if the administrator wishes so. The visitor could also personalize the museum to his interests and then view the model. For example, a musician visiting a medieval museum could choose to view only the medieval items which are relative to music (instruments, musicians' clothes etc.). His choices can be stored in a 'cookie' so he can view the same model later on.

7 Interoperability

VRML allows the creation of platform-independent 3D objects, described in text files, which can then be displayed on any computer platform for which an appropriate

browser exists or plugin. Since today, many VRML browsers have been created and for different platforms.[2] Some of these browsers are ported to a great number of popular platforms, like Windows, Linux, MacOSX, Java or Unix and it's variants (BSD, IRIX, Solaris etc).

Similar is the case with MySQL which has full server support for Windows XP/2003, MacOSX, Solaris, Linux, BSD, HP-UX, AIX, Netware, OpenServer, IRIX and others[3].

The most common PHP installation is probably the PHP module running with Apache on Linux or a UNIX-variant. But PHP also works on Windows NT and 9x, as well as with a number of other Web servers. You'll find more documentation floating around on the Web that's specific to the Apache/Linux/PHP combo, but it isn't by any means the only platform that PHP is supported on[4].

8 Acknowledgments

The current paper is supported by the SEEArchWeb: An Interactive Web-based Presentation of South-Eastern European Archaeology project - a SOCRATES programme, with grant agreement number 110665-CP-1-2003-1-MINERVA-M.

9 Conclusion

We have presented a Dynamically created Virtual Museum. Building a custom museum is now easy for every visitor of the museum or any museum executive, through an easy to use interface.

VRML was chosen primarily because it's an open, web-based protocol. Although there is a newer protocol available, called 'x3d', designed by the same team (w3c) as a replacement for VRML, we believe that VRML is more mature, with more tools and viewers available.

Although, the whole project was build using non-proprietary tools, a step forward to improving would be to support more databases and probably export both VRML and X3D models.

References

[1] *MySQL* database
http://www.mysql.com

[2] *VRML*, Web3D Consortium
http://www.web3d.org/

[3] Copyright 1997-2004, The *PHP* Group
http://www.php.net

[4] The VRML Pepository
http://www.cs.nchu.edu.tw/ tjsheen/martin /web3d/bk2.htm

[5] Real world use of XML, XSLT and Web Services in Archaeology
http://www.caaconference.org/

[6] Virtual views of the West House
http://www.caaconference.org/

[7] CAA2003 Ausserer, K.F., Bo"rner, W., Goriany, M. and Karlhuber-Vo"ckl, L. (eds) 2004. Enter the Past. The E-way into the four Dimensions of Cultural Heritage. CAA 2003, Computer Applications and Quantitative Methods in Archaeology.

[2] http://www.web3d.org/applications/tools/viewers_and_browsers
[3] http://www.mysql.com/support/supportedplatforms.html
[4] http://gr2.php.net/downloads.php

[8] CAA99 Fennema, K. and Kamermans, H. (eds) 2004. Making the connection to the Past CAA99. Computer Applications and Quantitative Methods in Archaeology.