Game Computational Programming

Assignment 1

Giannandrea Grone

Introduction

In 3D graphics there are many ways to represent the orientation of an object. The most common way is to use 3x3 rotation matrices or 3 Euler angles. However there is a third way of representing 3D rotation with Quaternions.

Design – Implementation

The implementation was designed to use both rotation Matrices and Quaternion to obtain the same behaviour and successively analyse the performance and qualities of each method:

The implementation is a simple model renderer in 3D space using OPENGL (glut), further mathematical methods permit rotation translation and some other 3D operations, and the code is divided in:

main.cpp where the rendering/initialisation and inputs are managed

"Quaternion.h" & "Quaternion.cpp” where the Quaternions are stored and all the useful operations are created

"MathHelper.h" & "MathHelper.h" where all other mathematical calculations are stored such as matrices operations.

"vertexShader.vsh" "fragmentShader.fsh" "dragon.h" there are also a fragment shader a vertex shader and a model which it is a dragon in this case.



Rotation Matrices implementation

rotXMatrix = new GLfloat[16];

In the implementation the rotation is stored in a 4X4 matrix, using GLfloats, since we are using OPENGL, in a pure mathematical example 3D rotation can be defined with just a 3X3 matrix.

In the mathhelper.cpp are defined multiple methods for matrices calculation in particular we want observe:

static void makeRotateX (GLfloat\* result, GLfloat rotation);

static void makeRotateY (GLfloat\* result, GLfloat rotation);

static void makeRotateZ (GLfloat\* result, GLfloat rotation);

static void matrixMult4x4 (GLfloat\* result, GLfloat\* m1, GLfloat\* m2);

The first 3 are used to modify the respectively values of a matrices to rotate it on a specific axis.

The last one is quite straight forward and simply multiply two 4x4 matrices, which is the main why to add rotation/scale and translation to a model.

Quaternions Rotation implementation

Quaternion(GLfloat w, GLfloat x, GLfloat y, GLfloat z);

A quaternion is defined by 4 values (WXYZ) in this case represented by 4 GLfloats, XYZ store the position and W the orientation. The most important methods are:

static Quaternion QuatRotate(Quaternion previous,GLfloat angle, GLfloat tx, GLfloat ty, GLfloat tz);

Used to rotate a quaternion by a given axis

static void QuatToMatrix(Quaternion& result, GLfloat\* matrix);

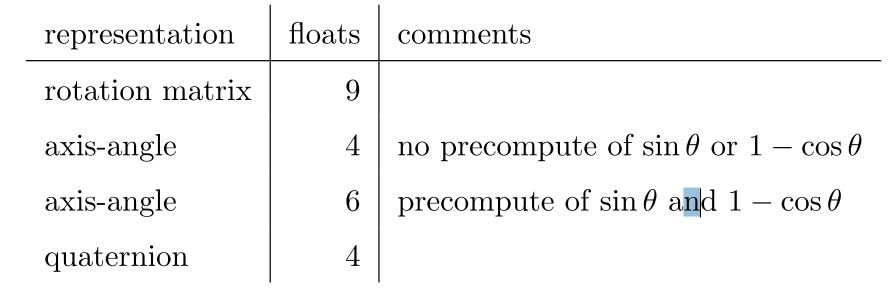
Which is necessary since opengl accepts only matrices

static Quaternion QuatMultiply(Quaternion a, Quaternion b);

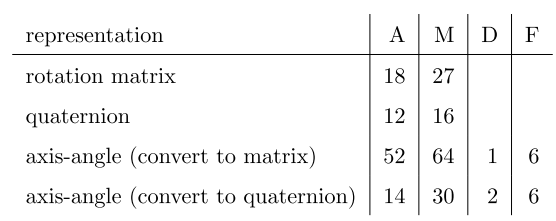
Which is useful to sum multiple rotations in a single Quaternion.

Evaluation and testing:

***Memory usage and computational time:***



As the table is showing, a rotation matrix requires 9 ﬂoats, a quaternion requires 4 ﬂoats, and an axis-angle pair requires 4 ﬂoats (or 6 if precomputed), undoubtedly the rotation matrix will use more memory and in most cases leaving the Quaternion as the cheapest of the three.



In regards of computational power required 2 rotation matrices multiplied toghether require 18 additions and 27 multiplication while 2 quaternions require only 12 additions and 16 multiplications. Making the quaternion the cheapest in processing power usage. However we need to keep in mind that opengl supports only rotation matrices and in the end we need to convert the Quaternion in a matrix.(see code below)

void Quaternion::QuatToMatrix(Quaternion& a, GLfloat\* matrix)

{

// First row

matrix[0] = 1.0f - 2.0f \* (a.y \* a.y + a.z \* a.z); // dotproduct

matrix[1] = 2.0f \* (a.x \* a.y - a.w \* a.z); // dot product

matrix[2] = 2.0f \* (a.x \* a.z + a.w \* a.y); // dot product

matrix[3] = 0.0f;

// Second row

matrix[4] = 2.0f \* (a.x \* a.y + a.w \* a.z); // dot product

matrix[5] = 1.0f - 2.0f \* (a.x \* a.x + a.z \* a.z); // dot product

matrix[6] = 2.0f \* (a.y \* a.z - a.w \* a.x); // dot product

matrix[7] = 0.0f;

// Third row

matrix[8] = 2.0f \* (a.x \* a.z - a.w \* a.y); // dot product

matrix[9] = 2.0f \* (a.y \* a.z + a.w \* a.x); // dot product

matrix[10] = 1.0f - 2.0f \* (a.x \* a.x + a.y \* a.y); // dot product

matrix[11] = 0.0f; // dot product

// Fourth row

matrix[12] = 0;

matrix[13] = 0;

matrix[14] = 0;

matrix[15] = 1.0f;

}

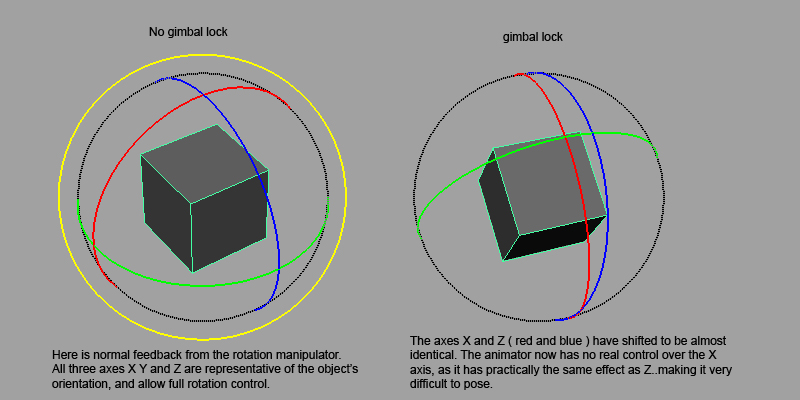
So every time we use a quaternion we need at LEAST to convert it once in a matrix using this algorithm which is formed by 27 multiplication 12 additions and 6 subtractions.

Versatility

***Quaternions advantages:***

**No ambiguity**, rotation on different axis can be performed in any order while Matrices and euler angles need to follow one of the six defined orders XYZ, XZY, YZX, YXZ, ZXY, ZYX.)

**No Gimbal Lock,** when using Euler angles, when 2 axis get close to be aligned, another problem rise up, the gimbal lock, which causes the lock or more precisely “ the loss of a degree of freedom” of one of the axis, breaking the rotation.



**Easy Interpolation,** with a quaternion it is just scalar multiplication and normalization, (with the help of SLERP). Expressing this with a matrix instead, requires evaluation of sin and cos, then building a rotation matrix.

**Less complicated, less round off error** as the quaternion requires less computation and especially less use of sin and cos functions, it’s more accurate in many cases

**One rotation, one quaternion:** quaternions represent one specific rotation while different angles can represent the same rotation (-180 and 180).

***Quaternions disadvantages:***

**Not as clear as Euler angles:** if you need to represent a single rotation such as a character orientation which doesn’t require more than rotation (Y), is probably more intuitive to use Euler instead of quaternions.

**CPU Friendly but not as much with Humans** difficult to understand and less intuitive to read for a human, the completely opposite for a computer which doesn’t need as much memory and calculations to get everything done.

**Evaluation and experiments**

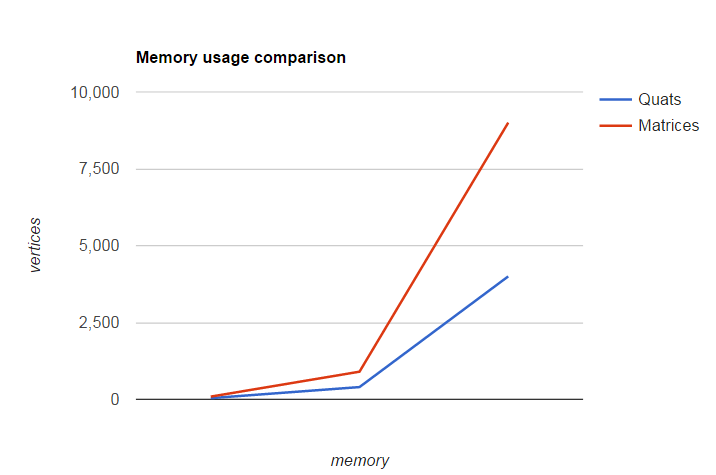
1. Memory usage

const int num\_indices = 300000;

const int num\_vertices = 49990;

const int num\_normals = 49990;

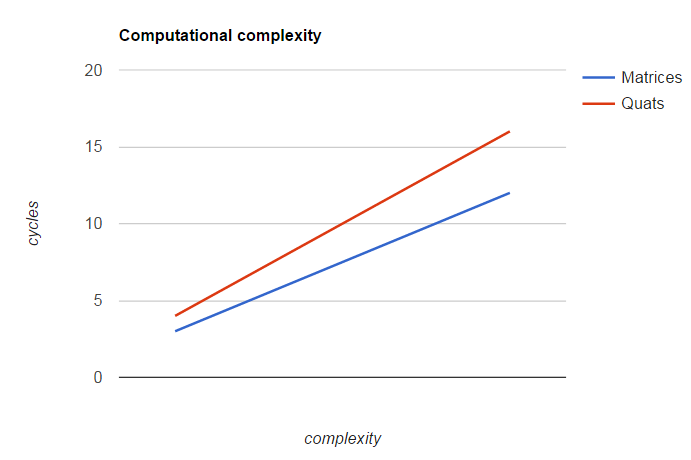
glBufferSubData(GL\_ARRAY\_BUFFER, 0, 3\*NUM\_VERTICES\*sizeof(GLfloat), vertices);

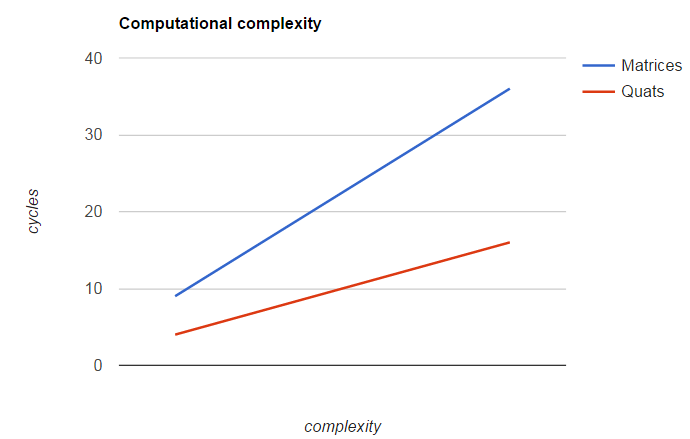


As we can see here the memory saved gets more significant as the number of vertices increase in case we use quaternions.

1. **Computational time and Algorithm complexity:**

http://puu.sh/swJAM/8805f186f2.png

On 1 axis

On 3 axis

As we can see as see in a 1 axis rotation matrices are cheaper but on a 3 axis rotation are significant more expensive compared to quaternions, even if we calculate the transform process of a quaternion becoming matrix for the rendering matrix.

Conclusion

In conclusion Quaternions seem more than simple alternative of matrices in 3D rotation, in most cases they seem more reliable and efficient for multiple tasks. Quaternions can handle significantly better extreme cases of rotation close to 0 values, such as at 0 degree and 180 degree (sin), and also they don’t suffer of the annoying gimbal lock.

Also quaternions seem more suitable for animation blending and interpolation since they guarantee a smoother transformation to the next animation, for example they would be perfect for a skeletal animation system for a human character, where one must evaluate a lot of translation/rotations for a large number of vertices.

However they are harder to visually understand and difficult to work with for the human eye, therefore I wouldn’t recommend them to be used in an artist software or at least be converted in Euler angles for the UI.

Use Quaternions for:

- Videogames

- Complex animation System

- When memory and computational power is an issue

- Complex simulation which require precision

Use matrices for:

- Simple rotations (2D or 1 axis rotation only).

- Visualisation of the actual angles

- in the UI (better converts quaternions in matrices before making artists cry!)