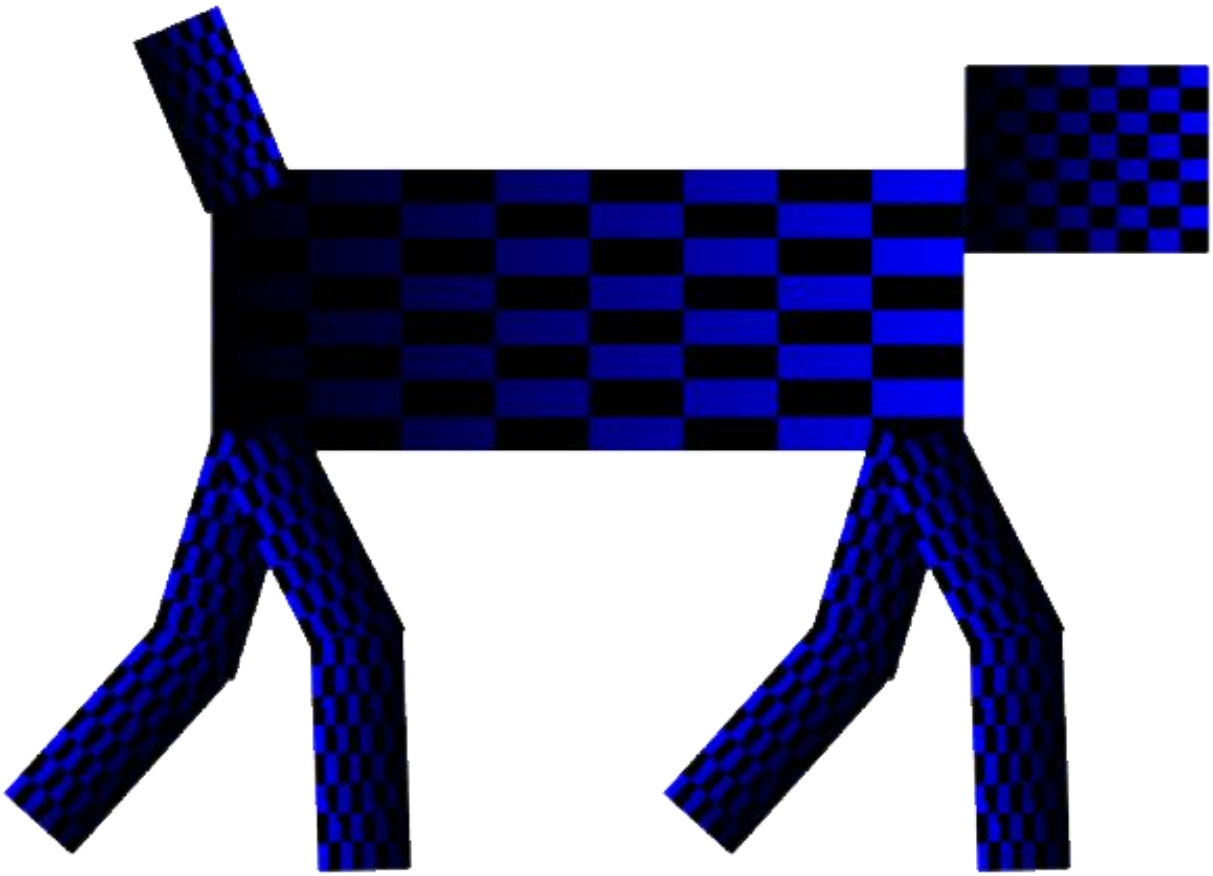
Interactive Graphics Report

Building, Texturing and Animating a Hierarchical Model of a Dog in WebGL



"LA SAPIENZA", UNIVERSITY OF ROME

3 giugno 2018

Autore: Simone Faricelli 1647406

INTERACTIVE GRAPHICS REPORT

BUILDING, TEXTURING AND ANIMATING A HIERARCHICAL MODEL OF A DOG IN WEBGL

HIERARCHICAL MODELS

After taking care of the various approaches to build and animate a single geometric object, in order to better represent an abstraction of the world, we are now focusing on a more complex model, composed by a set of the above-mentioned objects, linked each other by a *hierarchical relationship*.

THE TREE STRUCTURE

A way to visually represent the **Hierarchical Structure**, is through the use of a graph consisting of a set of *nodes*, which represent simple geometric objects, and a set of *directed edges*, which connect a pair of nodes in a certain direction. The particular graph that fits best for the purpose is the **Tree Graph**. In this model each node has an entering and leaving edge, linking it respectively to its *parent* and *child* node, with the exception for the nodes which has no entering edge, the *root nodes*, and that ones which has no leaving node, the *leaf nodes*, in addtion, we have no closed paths or loops.

FROM MODEL TO CODE

In order to express this structure in a graphics program, all the necessary information about the objects need to be stored inside the nodes themselves (al least):

- Object Drawing Function
- Transformation Matrix
- Pointer to the children node

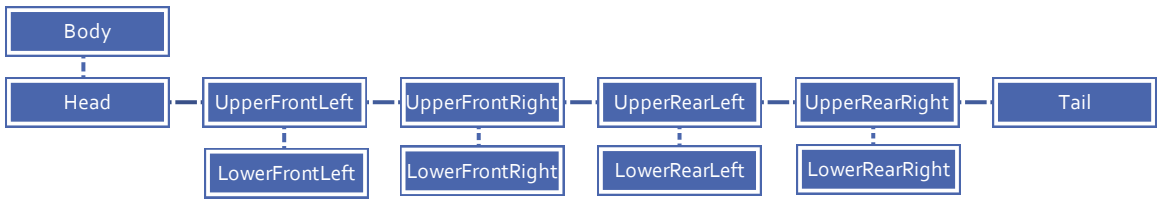
Needs to be said that each node's transformation matrix is applied on its parent's reference frame, so each matrix represents the incremental change when we go from the parent to the child. This happens because the structure is analysed through a process called **Tree Traversal**, in which every node is visited, traversing the tree from left to right, depth first, and at each node is computed its matrix.

TRAVERSAL ALGORITHMS

There are basically two different ways to implement the traversal algorithm.

In the first approach matrices and attributes of each node are *explicitly* stored into a stack and recovered when needed. The code is therefore well readable by the programmer who dealt with pushing and popping the information, but it is not extendable ore usable dynamically due to the fact that is built for the specific case. This tecnique is called **Stack-Based Traversal**.

The second approach, the one I used for this project, called **Preorder Traversal**, is undoubtedly a more general, which means independent of the particular tree, and powerful algorithm to implement traversal tree in which the storage of matrices and attributes is done implicitly. The image below represents the actually implemented tree structure for the building of the dog, in which vertical edges mean *Child* and horizontal edges mean *Sibling*.



IMPLEMENTATION

It can be easily understood from the graph above, the need to add inside the node structure, beside the information mentioned before, the data about the sibling, as it can be seen inside the function createNode(transform, render, sibling, child). As for the child, the sibling contains a pointer to the sibling node.

After initializing all the nodes, assigning an index, building each a drawing function and filling each one through initNodes(id), we can finally define the traversal tree function. What happens inside traverse(id) is as simple as powerful:

1. Graphic state (modelViewMatrix) is pushed inside the mvStack
2. Object of the node is drawn
3. Tree is traversed recursively:
    - *Child* starts from the previous state and applies its own transformation
    - Before applying the transformation for the *Sibling* the original state is Popped out of the mvStack

TEXTURE MAPPING

As discussed in the previous homework, the *Rasterization* process generates a set of fragments, which contains data about the pixel to which they correspond, then we must assign a colour or a shade to each fragment to proceed with the *Fragment Processing*. However, during this step the colour can be altered with an image or a pattern through the **Texture Mapping.**

The three basic steps needed for this process will now be analysed.

PATTERN GENERATION

The first step is about the building of the pattern and its loading into memory. Multi texturing is here needed in order to satisfy the requirements of checkboard, the first texture, and intensity gradient, second one.

At first an empty image of texSize*texSize resolution is created and filled by a double for loop with the checkboard pattern. Same for the gradient with a second empty image. It's then created a texture in webGL and bind it to an array for stacking in memory, respectively through the functions gl.createTexture() and gl.bindTexture(). Eventually we can deliver the data of the image to a two-dimensional array through gl.textImage2D().

TEXTURE COORDINATES

The key element in applying a texture in the fragment shader is the mapping between the location of a fragment and the corresponding location within the texture image where we will get the texture colour for that fragment. We could choose to explicitly identify the desired location in the texture image, through a *mathematical model* or some sort of *approximation*, and provide it to the shader, making the texture ad hoc for the model.

Nevertheless, in this project we are opting for letting the shader and the application to determine the appropriate texture coordinates for a fragment, making the code really more lightweight and simple. In order to achieve that aim, we're treating texture coordinates as a **vertex attribute**. Through the variable texCoord we're assigning *unitary* texture coordinate to the four corners of each face of parallelepipeds which build the entire model.

FRAGMENT SHADER

As made in the first homework for the vertex colour, texCoordArray is pushed into memory, passed to the *vertex shader* and rasterized with the identifier vTexCoord and finally multiplied to the *fragment colour*, inside the *fragment shader*, through the function texture2D and the variable type sampler2D, thich provide access to a texture object and all its parameters

ANIMATION

The process that brings to an animation is been fully discussed in the previous homework. Nevertheless that argumentation can be extended talking about the techniques about the management of frame rate.

INTERVAL AND TIMEOUT

There are basically two ways to make an animation smoother.

The first one consist in replacing the execution of the function render(), at the end of the loading's process of the page, with setInterval(render, 16), which cause the render to be called 60 times per second (every 16 milliseconds). However, it's possible not to achieve our goal because of the function's independence of the browser.

That's why it's been chosen to use the second approach for this homework, which consist in using the fully supported function requestAnimFrame(render) in combination with setTimeout(function() {...},100). The first one requests the browser to display the rendering the next time it wants to refresh the display and then call the render function recursively, while the second one allow to choose the frame rate.