HomeStyler 3D Logic Overview

This document tries to describe (as clearly as I can) the inner workings and the logic that is used in both iOS’s and Android’s HomeStyler apps. Please leave comments where further explanation is needed.

# Mathematical Requisites

As with any 3D application, some knowledge in Linear Algebra is encourage. Please review the following subjects:

* [Vector](http://en.wikipedia.org/wiki/Euclidean_vector#Basic_properties)
* [Matrix](http://en.wikipedia.org/wiki/Matrix_(mathematics))
* [Dot-product](http://en.wikipedia.org/wiki/Dot_product)
* [Cross-product](http://en.wikipedia.org/wiki/Cross_product)

Despite the above, most calculations could be understood in more simpler terms, so unless otherwise specified, skimming over the general definitions and properties of the above subjects would be enough to grasp the logic in this document.

# World, Camera and Entities

Both iOS and Android share the concepts of *entities* (the furnitures) that reside in the (3D) *world* that the user look at through a virtual *camera*. It’s crucial to understand what each of these terms mean (and don’t mean).

## Entity

An entity consists of a 3D mesh (the shape of the object), a texture (the paint of the shape), spatial properties consisting of position, rotations and scale, and other metadata describing abstract properties of the object (e.g. if it’s attached to the walls of the room).

Often times the concept of entity and model get mixed, and it’s important to separate the two[[1]](#footnote-0). The model is just the 3D mesh and texture, but it has no properties connecting it the world. For instance, while the coordinates of the vertices (points) that defines the shape of the model entails an internal scale of the object, that doesn’t (necessarily) corresponds to actual scale of that object when it’s placed as an entity in the world. The scale parameter of the entity translates the internal scale to the actual scale visible to the user.

### Mesh

The 3D mesh is (as mentioned above) the shape of the object. In our application, that data is represented in the Wavefront[[2]](#footnote-1) Object format, usually referred to as OBJ format. OBJ is a simple yet versatile format for describing 3d scenes. Its represented in a textual format consisting of lines of textual commands followed by several (usually) numerical parameters.

The basic commands that we use are the vertices commands (v, vt, vn), start of group command (g), and face command (f). The full format is available [here](http://www.martinreddy.net/gfx/3d/OBJ.spec).

The data that we care about when representing an object is the faces (in our case triangles) that make up the object. Each face consists of the vertices that define the face, the normals of the vertices that are perpendicular to the face (usually used for lighting), and vertices of the matching face in the texture image. Usually in our app a single object is composed of several sub-objects or groups of faces. That is used to separate the shadow plane from the actual furniture, for drawing the contour and for solving a weird issue that JPCT is having while drawing transparent objects.

It’s important to note that while OBJ files may reference textures and materials, it doesn’t contain such information in itself.

### Texture

Textures are image files that are used to “paint” the mesh of the model. The way this works is by taking a triangle cut of the image, and scaling and rotating it to fit the triangle in the 3D world. Textures are always a square that its sides are a power of 2 (due to hardware optimizations). The coordinates of textures are *normalized*, that means that each side is of length 1. The x-axis is going from left to right, and the y-axis is going from top to bottom. The origin is at the top left corner of the image.

A useful property of textures is that we can set the hardware to treat coordinates with values greater than 1, as wrapping around to its starting position. That means that (0.5, 0.5) and (1.5, 0.5) translates to the same pixel in the texture. We can use that to tile textures repeatedly on a single face. We use this trick in several places in the app (in drawing the wallpaper, cube wireframe and measuring tape).

#### Image representations

Because almost all of our furnitures has transparency in their texture (usually due to the shadow plane), we can’t use the JPEG format that doesn’t support transparency. Due to that need, we can either use PNGs, which provide for a lossless compression including the alpha channel, or provide our own solution to have transparency even with JPEGs. In most of the app we’ve opted for the later solution.

To represent a texture as JPEG, we separate the texture to two files, one containing just the color data that is saved as a JPEG, and one containing just the alpha in grayscale format saved as a PNG. In the client we combine the two images to a single complete texture.

### Metadata

Currently the metadata on objects is represented by a single numerical value called the z-index that is packed in a plist file (an xml file that Apple uses for all its platforms).

The possible values for the z-index are:

* 100 - Entities that are attached to the floor and shadows should be cast upon them (like carpets).
* 200 - Entities that are attached to the floor (tables, chairs, etc.).
* 300 - Entities that are free to moved in all 3D directions (vases, cups, televisions, etc.).
* 400 - Entities that are attached to walls (paintings, wall cabinets, etc.).
* 500 - Entities that are attached to the ceiling[[3]](#footnote-2) (lamps).

## World

The world is the virtual place where the entities resides, that is, entities are positioned in the world’s coordinates. The world in our case is the room that we place furnitures in. What needs to be understood about the world is that it has nothing to do with how objects appear to the user; that’s the function of the camera. I’ll go over that last point in the next section.

An important point is that Android’s JPCT is using a different coordinate system than the one used in iOS. iOS is using the standard OpenGL’s coordinate system: x-axis goes from left to right, y-axis goes from bottom to top, and the z-axis goes from front to back (front meaning where the user faces). JPCT’s coordinate system: x-axis goes from left to right, y-axis goes from **top** **to bottom**, z-axis goes from **back to front**. While there’s a difference between the coordinate systems that plays a part when loading/saving designs from different devices, almost all calculations are independent of the coordinate system used, and so I’ll be using throughout this document iOS’s coordinate system, and point out when adjustments should be taken for Android’s.

Implementation wise, on Android there’s an actual World class that handle all the above. On iOS this would be the Furniture3DViewController.

### Transforming Entities

If we simply draw an entity on the screen as is, it be placed at the origin of the world, at its internal scale facing to the user. In order to position the entity in the world we have to transform it from that initial state to its final position in the world; that transformation is done by multiplying points representing the entity by a matrix called the *model matrix*. In other words, the model matrix convert the model space to the world space.

The model matrix is actually composed of 3 matrices that either change the position, rotation or scale of the entity; to form the model matrix, these matrices are multiplied together. The order of multiplication of the matrices is important, and should be:

The reason the order is important is because the rotation and scale matrices operate under the assumption that the object is at the origin, and so should operate on the entity first. The scale also depends on the orientation of the entity, because the scale can be changed in each axis separately, and so it should operate before the rotation.

How each matrix is specify isn’t needed for understanding this document, as there are functions on both iOS and Android for creating them; those who are interested, can read about [homogeneous coordinates](http://en.wikipedia.org/wiki/Homogeneous_coordinates).

## Camera

The camera is how we view the world. It consists of two parts: 1) the position and orientation of the camera - the camera space; 2) the field of view, and aspect ratio of the screen - screen space. The first part is like the photographer holding the camera from a certain position, and pointing it to the direction of the subject of the photo; the second part is like the lens and film of the camera, that control how the world translates to a 2D image.

The first implication that should come to mind, from the fact that the world and camera are two separate objects, is that when the user tries to move the furnitures in a certain way, for instance into the back of the scene, that doesn’t necessarily means just changing the z position of the entity, because the coordinates of the camera don’t necessarily align with the coordinates of the world[[4]](#footnote-3).

As in the previous section, we use matrices to describe each part of the camera functions. The first part is called the *view matrix*, and is often times combined with the model matrix to form the *modelview matrix*. The view matrix translates the world space to the camera space. This part can be defined by:

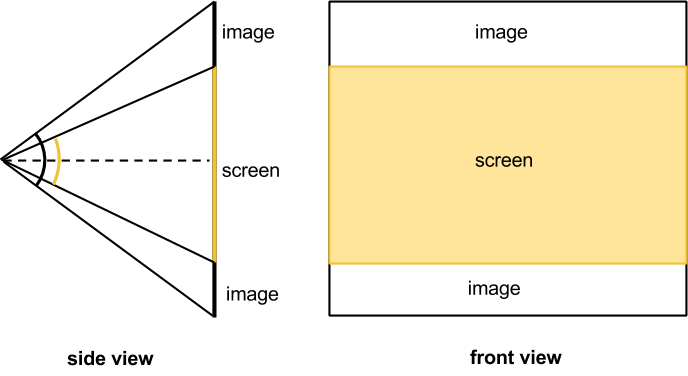
Position is (currently) just used to give the camera some height, which defaults to 1.5m above ground, but it can be changed by changing the floor position in the Real Scale screen. The rotation is the notorious rotation matrix that is passed around either from the gyroscope of the device, or from Adi’s 3D analysis algorithm in the server; this matrix is the own responsible for the orientation and direction of the camera.

The second part of the camera is called the projection matrix, which translates the camera space to screen space. This part is dictated by two main parameters: *FOV* (field of view) and *aspect ratio* of the screen; both will be discussed next. There are two more parameters called the near and far clipping planes, which aren’t necessary for the understanding of these document, but can be understood as the nearest and farthest distance from the camera that objects can still be seen; anything outsides that area isn’t drawn on the screen. On more note on the FOV is that there actually two FOV: the horizontal and vertical FOV. When both the horizontal and vertical FOV are known, the aspect ratio can be calculated directly (using simple trigonometry), and in general given two of the three values, the third value can be derived. In our app we usually work with the vertical FOV and the aspect ratio (due to the function creating the projection matrix).

### The camera’s function in the app

In our app, the camera has an important function, more so than in other apps. We need the virtual camera to match the camera that has taken the background picture, so that the 3D objects we draw would look like they belong inside the background image.

Assuming we know the orientation of the camera and the FOV of the lens, and the 3D view is the same size as the image, we can just input all those values to the native functions to create the projection and view matrices, and we’re set. The problem is when we want to look at part of the image, which happens when the screen size is of a different aspect ratio then the original image. In this case the FOV of the virtual camera needs to be adjusted to the different aspect ratio of the screen. As of now, when an image is too big for the screen, the image fills the screen so that either the width or the height of the image fits the screen (see the figure below). Since in our case we only care about the vertical FOV, the case where the aspect ratio is of the image is larger than the screen is trivial (the vertical FOV stays the same). In the case where the aspect ratio is smaller than the screen, we’ll have to adjust the FOV. As shown in the figure, it’s suffice to know the ratio between the normalized heights of the images to find the new FOV.



One more thing to note is that in older versions of the app, there was an error in the calculation of adjusting the FOV (my bad!). That’s why we need to due farther adjustments base on the design versions, the details of which I’ll leave for a separate document.

# Manipulating Entities Using Gestures

## General Note About Handling Gestures

The output of most gestures is usually a value that has no direction connection to the actual action that the gesture suppose to apply; this entails that we’ll usually need a constant to convert between the raw output of the gesture to the input of the logics of handling the gesture. These constants that are present in the code have no special meaning other than “they just work”, and so I won’t even mention them in this document, and instead would focus on the heuristics and logic part.

## Active Entity, Picking, and the Selection Gesture

The active entity, is the entity that all the gestures (other than the obvious selection gesture) act upon. In all gestures, if no active entity is currently selected, then the entity that’s behind the first touch of the gesture is selected. The active entity could also be set by a single tap on the desired entity as it appears on screen - this process is called picking.

Picking is the action of returning the entity that is visible in a specific point on the screen. There are several ways to accomplish this.

In the iOS app we use color picking. Color picking means, to draw the entire scene, such that each entity is drawn with a single constant and opaque color, that is uniquely associated with that entity. After all objects are drawn, we can sample the color at the desired point on the screen, check which entity is associated with that color. The entity class has a static long variable that is incremented each time an entity is created, and we use that as the unique identifier of the entity. To convert the identifier to RGB, we simply separate it to the component with bit shifting - the first 8 bits are red (R), and next 8 bits (9-16) are green (G), and the next 8 bits are blue (B). To convert it back is done similarly to compose the number from the components.

In the Android app, we use the built it picking mechanism, which as far as I can tell, uses Ray Picking, which means “firing” a ray from the pixel to the scene, and check which objects intersect with it, and take the nearest one.

## Two Finger Pan Gesture (Lifting entities up and down)

The simplest kind of interaction with entities is to move them up and down, i.e. changing the y position of the entities.

The naive method to implement the gesture is to just take the y-component of the translation of the gesture, and add it to the y component of the entity - there are several troubles with using this method. First, consider that some furnitures can’t be lifted, and that furnitures can’t be below the floor level. Second, consider that the pixel translation isn’t in the same “units” as the 3D world distances, so if you move your fingers one pixel up, and the world units are in meters, then you’re moving the object one meter upward - the solution is to use a “speed” constant to adjust for the difference. Third, consider that the screen size of the 3D view affects the calculation, which means that for larger views the entity would move slower (relative the screen size) - the solution is to normalize the translation vector with the size of the view (width for x position and height for y position); this would also affect the previous constant. And last, consider that the “up” direction, isn’t necessarily aligned with the frame of the device, because the picture could be taken in any angle - the solution would be to figure out the “up” direction with the accelerometer and take the component of the translation vector in the direction of the real “up” vector.

To expand on the last point:

* The real “up” vector can be found with the gravity vector - just take the (x,y) components of the 3D gravity vector, normalize it, and then invert it. Also take into consideration the case where you’re look almost directly at the floor (e.g. gravity.z > 0.95), in which case taking the naive up vector (x:0,y:1) would be just fine.
* Assuming u is the up vector, and t is the normalized translation vector, then dot\_product(u,t) would be the component of t in the direction of u.

## Rotation Gesture (rotating entities around themselves)

Rotation gestures are simple: given the rotation angle amount, just adjust the y rotation property of an entity. Just note that rotation has priority over scale, and so if rotation is started, then scaling is forbidden. If scaling has started then rotation is forbidden, but scaling is only activated after a relative scaling change threshold (>1.2 or <0.8), which ensures the priority of rotation.

## Scale Gesture (Making entities bigger/smaller)

Scaling are just as simple: just multiply the relative scale with the current scale of the entity.

## One Finger Pan Gesture (Moving entities around)[[5]](#footnote-4)

### Ray Intersection With Plane

It can be handy when moving entities to use the point of intersection between a ray cast from a point on screen, to a plane on which the entity is moving upon.

Let p0 be the start of the line, p1 the end of it, v0 a point on the plane and n the normal to the plane. Also, let u be equal to abs(p1 - p0).

The line equation is represented by .

A point p is on the plane iff .

If the line intersects the plane, then there’s a value such that is on a plane, which means that and so by simplifying the equation we’ll get:

Now if we assume that we can divide and solve this equation:

To understand our assumption better, the condition simply means that the line isn’t parallel to to the plane. There are two cases when a line is parallel to the plane: 1) the entire line is contained inside the plane, and so there are infinite points that intersects the plane, in which case we’ll usually take p0 to be our chosen point of intersection; 2) The line is parallel but not isn’t contained in the plane, and so there are no intersection points.

To summarize our calculation, **assuming the line is intersecting the plane**, the solution is:

If then return , else return .

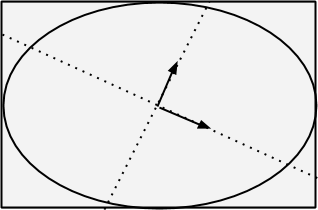
### Movement on Floor

There are two ways to implement the movement of entities on the floor; it could be done by using ray intersection or by using heuristics of converting the movement on the screen to a movements on the floor.

Using ray intersection is in many ways very simple: just move the entity to the point on the floor the user is touching. One slight modification of that method would be to allow for relative movements of entities even if not touching them directly; to accomplish that, instead of directly moving the entity to the point of intersection, we can keep just add to the entity’s position the change between the intersection point of the start of the gesture, and the intersection point of the end of the gesture - this way if the user starts by dragging on the entity directly, the entity would move to where the finger is at, but still allows for relative movements when not touching the entity directly. Another improvement would be to figure which plane is best used for finding the intersection point; while using the plane that the pivot of the entity is on, is a good starting point, a better one would be to divide the entity to several planes along the y-axis, and find on which plane the distance between the intersection point and the entity’s center is the closest (basically means where the line passes closest to the entity) - this would allow for better handling of simulating “grabbing” the entity.

The issue with the ray intersection method, and the reason why it’s currently turned off in the iOS project (though it’s fully implemented), is that in some cases ray intersection might be unintuitive to the user. One such instance is when the user starts dragging his hand from the bottom of the screen to the top, passing above the “horizon” of the plane, causing the entity to go to infinity. Another issue is that for entities that are already “above” the camera, movement is reversed, meaning that moving the finger down, causes the entity to move forward instead of backward. All these issue might have workable solutions, but they are at the UX level, not the logical level, and should be consulted with the UX designer.

The heuristic method is composed of several heuristics:

1. Like the lifting gesture, the translation vector is decomposed to an x and y components that are relative to the orientation of the device.
2. Like the lifting gesture, we’re trying to normalize the translation vector relative to the size of the view. Normalizing the translation vector according to it’s natural axis didn’t prove good enough, and so I tried to normalized it along the orienten axis - this means normalizing according to the length of the line that fits on screen - see the following figure:To more easily calculated those lengths, I approximated the rectangle with an ellipse.
3. Like in the lift gesture, the pitch of the camera affect the movement of entities. Consider that the smaller the pitch, there’s more depth packed into the bottom part of the screen; as you look down, that “depth” is spread across more of the screen height until you look directly at the floor and it encompassed the entire screen. This suggests that if the pitch is small, the xz-movement should be faster, and if it’s large, the movement should be slower. To account for this, we use the z component of the gravity vector (relative to landscape) as a measure of the pitch. The higher the length of the z-component, we’re looking more towards the floor, and so the pitch is larger. Since the gravity vector is normalized, the component is in the range [0,1], and so we use the the mapping , so if then pitchFactor is 1 and so the movement is normal, but if then pitchFactor is 0.5 and so the movement is slowed by half.
4. Entities which are farther away should move faster, because the changes in movement appear smaller in the distance. This means we should use the z-position of the entity as the factor (with a suitable conversion constant); but please pay attention that when the object is near, the z-component goes to zero and so the movement becomes too slow, and so we put a lower bound of 1 on that factor.
5. Movement along the z-component should be faster than the x-component, because the z-component looks more “dense” from the perspective of the user.

The exact constants mentioned in the above heuristics are almost always never given explicitly, because they have no special meaning other than “they work”, and they could be found at the code. The intent is to make some sense of the formula, while keeping in mind that many other good sounding ideas to make the movement behave nicely have been tried, and sadly failed.

### Wall Entities

The wall movement can be divided to two parts: 1) movement on the same wall, and 2) movement between walls.

The first part is quite simple; use the ray intersection method mentioned above, but take the plane to be the wall on which the entity is on. The problems with using ray intersection for entities on the floor, aren’t present for entities on walls, and so it’s better than finding special heuristics. As long as the entity is within the bounds of the wall, everything is quite simple. If the entity tries to move beyond the ceiling/floor then eliminate the movement along the y-component to still allow the entity to move along the xz-plane while being stuck along the y-component.

The second part is a bit uglier[[6]](#footnote-5). First when need a way to know when an entity is near the boundaries of wall it’s currently on, which means we first need to know on which wall it is on right now. In theory that should be easy, just take the distance between each of the walls, and look for the wall most close to the entity. This works well up until the entities starts to get near the corners of the wall, where this logic doesn’t work as good as one might expect[[7]](#footnote-6). The solution I’ve found (that still doesn’t always works), is to use the distance method on for the start of motion, and then rely on the first result for checking the boundaries of the wall. The method that accomplish just that is called “wallAtPosition…”. It returns the current wall the entity should be on, the direction of the x-component of the movement, and the normal to the wall. These parameters helps move the entity correctly along the wall, and in case of switching walls, it helps figuring how to rotate the entity to face in the direction of the wall. One of the issues I’ve stumble is that when the entity is trying to pass both the ceiling/floor and to another adjacent wall at once, which makes determining the next wall harder for the wallAtPosition function. A more correct solution would be to add a wall property (or more generally parent-child relationship) to entities, and have the wallAtPosition use that property, and that it should also return all the boundaries that the entity is “trying to break through”, to help fix the issues mentioned above.

One should also note that in case that wallAtPosition can’t figure on which of the walls the entity is on, we reset the position of the entity to one of the corners.

Another issue that needs handling is the case where the walls have moved due to using the Real Scale feature - right now it’s handled by the above edge case of wallAtPosition not knowing on which wall the entity is on.

# First State of Newly Added Entities

When adding a new entity to the world, the following rules determine its initial state, segemented by the entity’s Z-Index:

* 100: The rotation and (x,z) position is the same as 200 (see below), but the y-coordinate is slightly below zero so that other furnitures could cast shadows upon it. Also, to fix issues with multiple carpets at the same position and the same height, we also add a small random variation to the y-coordinate.
* 200, 300: The y position is set on the floor, the (x,z) position is determined by ray intersection, where the ray is being cast from the point on the screen where x is the center and y is ¼ of the screen - this means all newly added furnitures will always be places at the center of the scene close to the user and easily manipulated to their right place. Also, the entity is rotates so it’s aligned with the cube of the room.
* 400: Places the entity at half the height of the visible portion of the room, near one of the visible corners of the room, or if both of them are hidden, then in the middle of the front wall.
* 500: using ray intersection, places the entity on the ¾ height of the screen similar to 200/300.

# Saved Design Format

I’m not going to go into the specifics of the format (which is a json dictionary), but about the reasons and changes between the versions of the format. It’s enough to know that the format consist of an unordered list of entities (position, rotation and scale) , the global scale, the FOV of the original photo, the room cube scale, a representation of the rotation matrix, and the format’s version; other metadata that’s part of the format is irrelevant to this document.

## Changes between version 1 to version 1.1

On the app where version 1 was the default, I made a mistake that entities that weren’t attached to the floor, had the wrong position relative to the floor. This was because it some point I switched from storing the absolute position of furnitures, to storing the position relative to the floor, and forgot to make to correct changes in some of the places. After I’ve fixed the error in the code, I had to fix the designs that have already been uploaded, to separate the wrong and the correct designs, I’ve increased the format’s version to 1.1.

To upgrade version 1 to version 1.1, there several measures that need to be made. First, it’s important to note that the goal is to keep the same appearance of the old designs, on expanse of the correctness of the design; to be more specific, I distort the cube to make the old designs work under the new interpretation of the same data.

1. For every object which isn’t attached to the floor, subtract the floor height from it.
2. Double the floor height of the cube by changing the bottom vertices of the cube (if it was lower than the zero xz-plane, then it would be twice as low; if it was higher, then it would be twice as high).

## Changes between version 1.1 to version 1.2

On the app where the format version was 1.1 and below, I made a mistake in the calculation of fitting the FOV of the original photo, to the visible part of the photo on the screen. The wrong part of the calculation was, that in the case where the photo was wider than the screen, instead of keeping the same vertical FOV, I’ve applied the following calculation:

the case where the photo is narrower than the screen is handled correctly by the old code.

The current solution is to read the FOV, apply the old wrong calculations, and then modify it to the different screens. This works because version 1.2 was introduced as part of the Android and iPhone versions of the app, and so the wrong calculation was only done on the iPad, and so it’s easy to keep the look of the design, even if we don’t display the right FOV fitting the image. While there’s a way to upgrade 1.1 to 1.2, we chose not to because there were many users using the old iPad version that still had the bug; we didn’t want to distort their designs. Instead, we’re reading the bad version correctly preserving the appearance, but saving it the same way as version 1.1 did. At some point in the future it would be wise to upgrade the old versions to the latest one, once the usage of the old versions would drop.

You can see the exact solution in the code, but to summarize:

1. If the format version is >1.1 then apply the correct calculations as described in page 5.
2. Else if the photo is narrower than the iPad screen (4:3 ratio), then again apply the correct calculations (as was always done).
3. Else (the photo is wider than 4:3 ratio), then apply first the wrong calculations as described above for a 4:3 ratio, and then apply the correct calculations (which depends on whether the photo is wider/narrower than the actual device’s screen (not necessarily an iPad).

# Multiple “Rotation Matrices” in the App - WIP

The rotation matrix is the matrix that specify how the camera is rotated, so it would be possible to place 3D entities in the right perspective so the entities looks like they are part of the image. Where things get complicated is that there are several representations of the same rotation matrix used in the app. Though all of these matrices represent the same data (how the camera was held), to interpret it correctly we must know the coordinate system in which it is represented.

* The OpenGL/iOS standard version: the rotation matrix is specified in the standard coordinate system, x goes from left to right, y goes from bottom to top and z goes into the screen. The rotation represents the rotation applied to the world in that coordinate system. This is the rotation matrix used in all the 3D parts of the iOS app. This rotation matrix is also relative to the image orientation, not the device.
* The iOS gyro version: The gyro in iOS is
* The JSON design format version: s
* The Android version: The coordinate system in JPCT is similar to the standard system, only it’s rotated 180˚ around the x-axis, which means that x goes from left to right, y goes from top to bottom, and z goes out of the screen. Due to fact that I had problems understanding exactly how jpct works, I’ve dropped using a rotation matrix to define the camera, and instead opted to using directional vectors: one specifying the direction of the camera, and an “up” vector specifying the rotation around the direction vector.
* The 3D analysis API version: the coordinate system is supposedly like the android version, but for some reason it didn’t worked right, until I’ve simply stumble upon the correct transformation between the standard version to the 3D analysis’s version.

**To Be Continued...**

1. To make things more complicated, while in the iOS app an entity is defined by the class Entity and its instances are always names as such, in the Android app, the class is named Object3D (part of JPCT), but its instances are always named model. [↑](#footnote-ref-0)
2. A funny anecdote: Autodesk currently owns what used to be Wavefront. [↑](#footnote-ref-1)
3. Though in our app we don’t actually restrict their movement. We just place them in the ceiling, but the user can still move them in all directions. [↑](#footnote-ref-2)
4. Though in the present implementation I make sure that the zy plane is aligned, so pushing the object back actually does mean changing the z position [↑](#footnote-ref-3)
5. Caution: this is the tough part... [↑](#footnote-ref-4)
6. It shouldn’t be like that, but I never had the time to go back and fix it properly. It’s a good place to find a better way to do this. Nevertheless, I’ll present the current implementation. [↑](#footnote-ref-5)
7. Though it might have something to do to the constant I chose for boundaries. [↑](#footnote-ref-6)