UNIVERSITY OF Hull

**DEPARTMENT OF COMPUTER SCIENCE**
**ASSESSMENT DESCRIPTION 2016/17**
**(EXAM TESTS WORTH ≤15% AND COURSEWORK)**

**MODULE DETAILS:**

| | | | |
|---|---|---|---|
| Module Number: | 08347 | Trimester: | 1 |
| Module Title: | Virtual Environments | | |
| Lecturer: | HW | | |

**COURSEWORK DETAILS:**

| | | | | | |
|---|---|---|---|---|---|
| Assessment Number: | 1 | of | 2 | | |
| Title of Assessment: | Individual implementation of a simple virtual environment using HIVE technologies | | | | |
| Format: | Program | Demonstration | Practical test | | |
| Method of Working: | Individual | | | | |
| Workload Guidance: | Typically, you should expect to spend between | 35 | and | 40 | hours on this assessment |
| Length of Submission: | This assessment should be **no** more than: *(over length submissions **will be** penalised as per University policy)* | N/A - coding exercise | | | |

**PUBLICATION:**

| | |
|---|---|
| Date of issue: | 3 October 2016 |

**SUBMISSION:**

| | | | | |
|---|---|---|---|---|
| ONE copy of this assessment should be handed in via: | Canvas | | If Other (state method) | Complete practical test at final lab 1 Nov 2016 |
| Time and date for submission: | **Time** | 9AM - 1PM | **Date** | 1 November 2016 |
| If **multiple hand–ins** please provide details*:* | Demonstrations during labs – see below for marks | | | |
| Will submission be scanned via TurnitinUK? | No | If submission is via TurnitinUK students MUST only submit Word, RTF or PDF files. Students MUST NOT submit ZIP or other archive formats. Students are reminded they can **ONLY** submit **ONE** file and must ensure they upload the correct file | | |
| | | | | |

The assessment must be submitted **no later** than the time and date shown above, unless an extension has been authorised on a *Request for an Extension for an Assessment* form which is available from: http://www2.hull.ac.uk/student/registryservices/currentstudents/usefulforms.aspx

If submission is via TurnitinUK within Canvas staff must set resubmission as standard, allowing students to resubmit their work, though only the last assessment submitted will be marked and if

submitted after the coursework deadline late penalties will be applied.

**MARKING:**

| | |
|---|---|
| Marking will be by: | Student Name |

**ASSESSMENT:**

| | | | | |
|---|---|---|---|---|
| The assessment is marked out of: | 30 | and is worth | 30 | % of the module marks |
| **N.B** If multiple hand-ins please indicate the marks and % apportioned to each stage above (i.e. Stage 1 – 50, Stage 2 – 50). It is these marks that will be presented to the exam board. | | | | |

**ASSESSMENT STRATEGY AND LEARNING OUTCOMES:**
The overall assessment strategy is designed to evaluate the student's achievement of the module learning outcomes, and is subdivided as follows:

| LO | Learning Outcome | Method of Assessment *{e.g. report, demo}* |
|---|---|---|
| ***3*** | *Implement a virtual environment using technologies applicable to the problem domain* | Demos, practical test |

| Assessment Criteria | Contributes to Learning Outcome | Mark |
|---|---|---|
| Demonstration 1 | 3 | 4 |
| Demonstration 2 | 3 | 5 |
| Demonstration 3 | 3 | 1 |
| Demonstration 4 | 3 | 6 |
| Demonstration 5 | 3 | 6 |
| Practical test | 3 | 8 |

**FEEDBACK**

| | | | |
|---|---|---|---|
| Feedback will be given via: | Verbal (via demonstration) | Feedback will be given via: | Mark sheet |
| Exemption (staff to explain why) | | | |
| Feedback will be provided no later than 4 'teaching weeks' after the submission date. | | | |

This assessment is set in the context of the learning outcomes for the module and does not by itself constitute a definitive specification of the assessment. If you are in any doubt as to the relationship between what you have been asked to do and the module content you should take this matter up with the member of staff who set the assessment as soon as possible.

You are advised to read the **NOTES** regarding late penalties, over-length assignments, unfair means and quality assurance in your student handbook, also available on Canvas – https://canvas.hull.ac.uk/courses/17835/files/folder/Student-Handbooks-and-Guides.

In particular, please be aware that:

- Your work has a 10% penalty applied if submitted up to 24 hours late
- Your work has a 10% penalty applied and is capped to 40 (50 for level 7 modules) if submitted more than 24 hours late and up to and including 7 days after the deadline
- Your work will be awarded zero if submitted more than 7 days after the published deadline.
- The overlength penalty applies to your written report (which includes bullet points, and lists of text you have disguised as a table. It does not include contents page, graphs, data tables and appendices). Your mark will be awarded zero if you exceed the word count by

more than 10%.

Please be reminded that you are responsible for reading the University Code of Practice on the use of Unfair means (http://www2.hull.ac.uk/student/studenthandbook/academic/unfairmeans.aspx) and must understand that unfair means is defined as any conduct by a candidate which may gain an illegitimate advantage or benefit for him/herself or another which may create a disadvantage or loss for another. You must therefore be certain that the work you are submitting contains no section copied in whole or in part from any other source unless where explicitly acknowledged by means of proper citation. In addition, **please note** that if one student gives their solution to another student who submits it as their own work, **BOTH** students are breaking the unfair means regulations, and will be investigated.

In case of any subsequent dispute, query, or appeal regarding your coursework, you are reminded that it is your responsibility, not the Department's, to produce the assignment in question.

# 08347 VE, ACW1 2016-17: Individual implementation of a simple virtual environment using HIVE technologies READ THIS PAGE

## Background and Approach

This piece of work takes a tutorial approach to understanding some key concepts in graphics, like projections, scenes and axial systems, and applies these in order to code a head-tracked, stereo virtual environment framework. The test scene is very simple – just a wireframe cube – but once these principles are mastered you will have the correct basis for *any* projection-based VE. Assessment is via demonstrations ongoing in the five two-hour labs, and a simple quiz at the final lab that ensures you as an individual understand what you are doing. See the last page for a sample quiz and the rules. Key technical points about the ACW will be explained in labs by the lecturers as we go along and there is also a summary of what to do in the code itself, in the form of comments. There will be opportunities to try programs out in the Green Room near to the Cray Lab, which has an identical tracking system to HIVE. Finally, you have to do the assessments in a timely fashion in order to be allowed to attempt the final quiz – see the Canvas assessment item 'Promptly Points' for more details, and the section below entitled 'Individual Canvas quiz'.

## Materials and Acknowledgments

Download the sandbox from the module web site and unpack it. The contents consist of a program constructed for ease and transparency rather than style and it is therefore **not** for distribution outside the University of Hull. The sandbox also includes a well-known open-source utility called VRPN that has been modified specifically for use in this piece of work, which must therefore also not be redistributed. Other essential open-source utilities gratefully acknowledged in building this course work are the Quat mathematics library (usually distributed with VRPN), the GLEW utility and the FreeGLUT library: consult the web sites of each of these if you want to know more about these great offerings and possibly use them in your project.

## Scope of the Work

You should only change the code in the routines ViewSystem, OnDisplay and the small, flagged part of the global section at the top. You can (and probably should) look at the other parts but don't change **anything** elsewhere, and don't worry if you don't understand it. Close up the irrelevant parts in the Visual Studio solution explorer once you've had a look, to make navigating the source easier. The steps to success are highly constrained and essentially require no programming skill. Add as little as possible to achieve the marks and, in particular, you should not need to make any new variables – see the instructions and ask if you are unsure. Note that you cannot revisit an assessment – only ask for a stage to be marked if you are sure you want to go ahead.

## Getting Started

The sandbox is complete and you do not need any other libraries, DLLs, toolkits etc. If any of the following steps do not work, as module staff immediately. Double-click the 'server_emulator' to emit positions (you might need to 'Cancel' a dialogue from the firewall), double-click the solution file (.sln extension) to open the Visual Studio solution, and click the green arrow to build and then run the program. That's it, it's working. The only other thing you'll need is your student card, to represent you in the assessment/help queues.

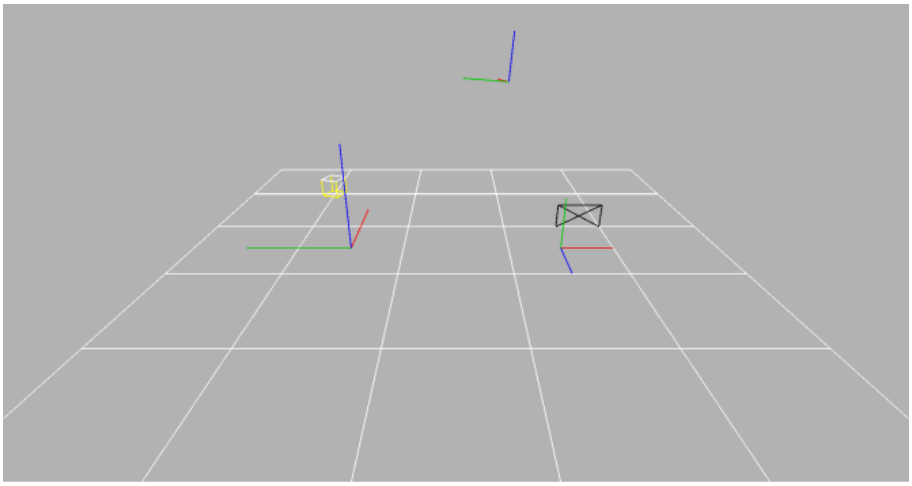
*[Note for future use*: If you want you can run the built program outside Visual Studio. First, start the emulator as above and then navigate using your Windows Explorer or MyComputer to the Debug or Release folder, whichever one you just built. In this folder, do SHIFT-RIGHTMOUSE and choose 'Open command window here'. In the command window, type 'set PATH=%PATH%;..\bin' *without* the enclosing ''. The executable file name is 08347ACW.exe so the command to run it is '08347ACW Glasses01@localhost' – again, without the enclosing '', and that's 'zero-one' after 'Glasses' and it's all case-sensitive. Re-running it is just UPARROW.]

## First stage – Familiarisation with Tracker and OpenGL axial systems

When it first runs the output shows moving axes following the glasses' tracked position and orientation. These axes and the large set on the ground plane are tracker objects drawn in a coordinate system whose origin is in the centre of the ground plane and whose X (red) axis is pointing away from us, Y (green) is pointing left and Z (blue) is pointing up. The OpenGL rendering system, on the other hand, follows the small set of axes on the ground. In this space X points to the right, Y points upwards and Z towards us. The other objects sitting on the ground plane are a monitor and a cube, constructed using the tracker coordinate system. This first stage of the assessment is designed to make you familiar with these two axial systems and how to specify and parametrise object transformations (scaling, rotation and translation) using either system.

**Change** the translations of the large and small stationary sets of axes so their origins lie on the tracker's Y axis, with the tracker axes set in the centre of the second ground tile from the left and the OpenGL axes set in the centre of the fourth tile. You will need to work in both tracker and OpenGL space to do this. Use numerical values rather than program variables for this part.

**Change** the monitor using the scale routine and its height variable, set to the same measured size as the window the graphics draws in on your screen, minus borders. Choose a numerical value for the cube size that will fit in this window and scale it too. Translate the monitor so its bottom right corner sits above the intersection of four tiles on the right, and translate the cube so its centre sits above the intersection on the left. All stationary objects must sit on, rather than penetrate, the ground and, furthermore, the cube must have a rotation, dependent on its face size, of 10° per 10cm. This must be clockwise when the cube's (i.e. the tracker's) Z axis points towards us. Parametrise translations using the object sizing variables plus 'viewaspect' if needed. **Do not make any new variables,** nor use 'screenup' or 'screendist' for now. Use numerical values if you are not given a sizing variable with which to parametrise a calculation. The first deliverable looks like this but use your own sizes for the monitor and cube:



**Assessment by program demonstration recorded as a Canvas quiz [4 marks]**

1. Tracker axes correctly orientated and positioned as required ('i'/'o' keys) **[1]**
2. OpenGL axes correctly orientated and positioned as required, and the tracker Y and OpenGL X axes must lie in the same straight line ('i'/'o' keys) **[1]**
3. Monitor maintains the same aspect ratio and remains with its corner on the ground on the specified intersection, as its height is increased and decreased ('+'/'-' keys) **[1]**
4. Cube remains with its face on the ground centred above the specified intersection and its rotation increasing and decreasing, as its size is increased and decreased ('+'/'-' keys) **[1]**

## Second stage – Move the monitor and cube onto the wall, look 'side on'

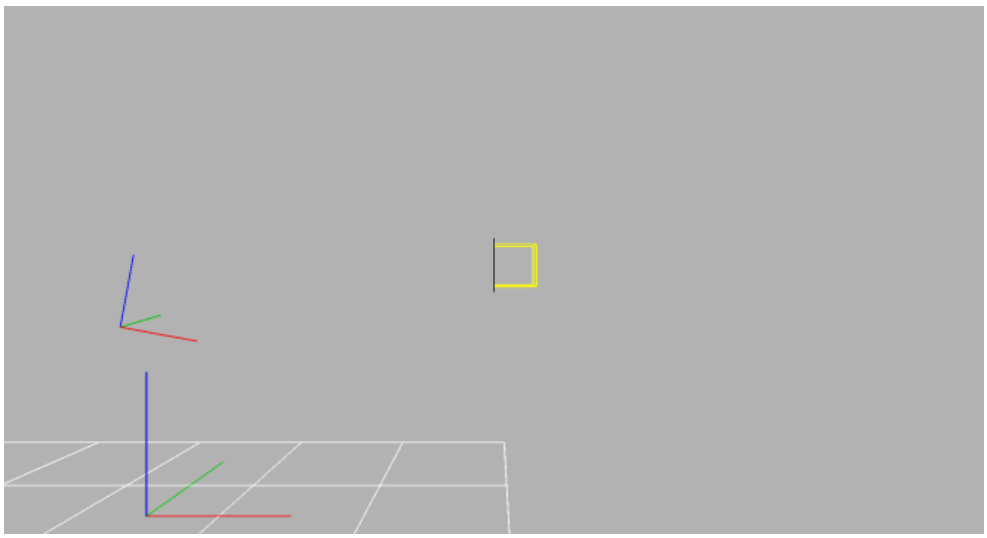
This stage is crucial because the white cube face on the monitor will become a stationary feature when the viewpoint is moved onto the glasses. This allows checking the frustum calculations.

**Rotate** the cube so its white face is towards us in the 'in front' viewing position ('i' key). Use numerical values rather than program variables for this part.

**Choose** a 'screendist' numerical value that will fix the monitor on a virtual wall a short distance beyond where the glasses-wearer walks. Check the oblique view ('o' key) to be sure the user will not walk through this wall, even at their furthest point forwards. You will sometimes use this distance in the negative sense from the origin but it is less confusing to specify a positive number here and put a negative sign in front of the variable when you need to. Specify a 'screenup' value that will put the monitor approximately opposite the user's head as they walk forwards. The variable 'screenup' is defined as the distance from the ground to the bottom edge of the monitor, not its centre. Now translate the monitor and cube onto the wall using these variables and those you set at stage 1. The cube's white face must lie at all times in the plane of the monitor and be centred with respect to it, and the monitor must be centred in front of the user.

**Fill out** the 's' camera preset ('side-on') in 'ViewSystem' so that the eye is in the same plane as the monitor and the cube's white face, looking from roughly eye height above the right-hand edge of the ground plane as seen from in front. The eye should look directly across the scene at a mirror of itself in the OpenGL X=0 plane. Leave 'up' pointing in the OpenGL Y direction.

**Do not make any new variables.** Use numerical values if you are not given a variable with which to parametrise a calculation. The second deliverable looks like this but at your chosen distance and upward displacement – notice there is no parallax on the cube's white face contained within the monitor but there is some visible on its rear face:

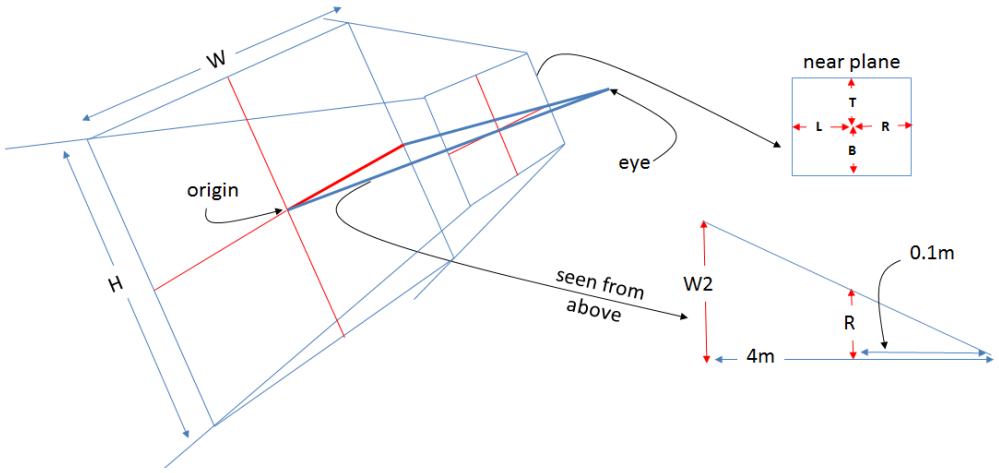


**Assessment by program demonstration recorded as a Canvas quiz [5 marks]**

1. Cube/monitor maintain initial configuration
    a. as their respective sizes are increased/decreased ('+'/'-' keys) **[1]**
    b. as they dolly back and forth ('uparrow'/'downarrow' keys) **[1]**
    c. as they move up and down the wall ('leftarrow'/'rightarrow' keys) **[1]**
2. Cube white face is embedded centrally in monitor **[1]**
3. Each new 'side-on' view ('s' key) exhibits required parallax/lack of parallax as the cube/monitor dolly back and forth **[1]**

## Third stage – produce a symmetric frustum using AsymmetricFrustum

This stage helps understanding the parameters of the asymmetric frustum, that is, the tapering shape used to view the scene, which gives a sense of perspective by making distant objects draw smaller than near ones. The aim initially is to roughly reproduce the look when using the routine SymmetricFrustum, by using numerical values in the slightly more complicated routine AsymmetricFrustum. Later we will substitute the calculations that we need for head-tracking and stereo. The numerical values we need are 'left', 'right', 'bottom' and 'top' in the diagram below (L, R, B, T for short, on the frustum's near plane) which can be estimated quite easily by thinking about the geometry of the scene being drawn:




**Estimate** the horizontal half-width of the scene in metres as it is drawn using the SymmetricFrustum routine, along the line running through the origin in the centre of the screen, using the one-metre tiles of the ground plane. Call this W2, to remind us it is the half-width. It is important to measure all of the scene that is drawn, even beyond the edges of the ground plane, to the very edge of the window. We do this at the origin because we set this at 4.0m from the eye in the 'in front' view and it gives us two sides of the emboldened right-angled triangle in the diagram above. Additionally, we know the distance from the eye to the near plane is 0.1m because we set it to be this. We can now use the ratios of sides in the similar triangles, giving the equations $\frac{L}{0.1} = \frac{W2}{4.0}$ and $\frac{R}{0.1} = \frac{W2}{4.0}$. A simple rearrangement will give L and R because we know W2.

**Estimate** the vertical half-height of the scene in metres, along the line running through the origin in the centre of the screen, using the one-metre-long blue arm of the tracker axes (the large set) sitting on the ground plane. Call this H2, to remind us it is the half-height. Using similar logic to that used in the L and R calculation we can get B and T from $\frac{B}{0.1} = \frac{H2}{4.0}$ and $\frac{T}{0.1} = \frac{H2}{4.0}$.

We're not quite done, though. The quantities 'left', 'right', 'bottom' and 'top' are not just sizes, but directions too. We can think of 'left' and 'right' as the x coordinates of the vertical edges of the near plane, and 'bottom' and 'top' as the y coordinates of the horizontal edges. This is OpenGL space, so 'left' is in the negative x direction and 'bottom' is in the negative y direction – put the **correctly signed** numbers into 'AsymmetricFrustum' (ViewSystem, 'bInperson' if block) and this stage is complete. Don't waste time trying to get your deliverable to look exactly like the original, but do check with the debugger that 'viewaspect' in your running program is roughly W2/H2. Also don't bother with tangents of the field-of-view and variables – this is a one-off calculation using the simple method above, to check you understand the principles. We'll also use similar triangles again later.

**Assessment by program demonstration (Canvas quiz) [1 mark]:** Using AsymmetricFrustum, scene renders symmetrically on camera preset 'p' with proportions similar to preset 'i' **[1]**

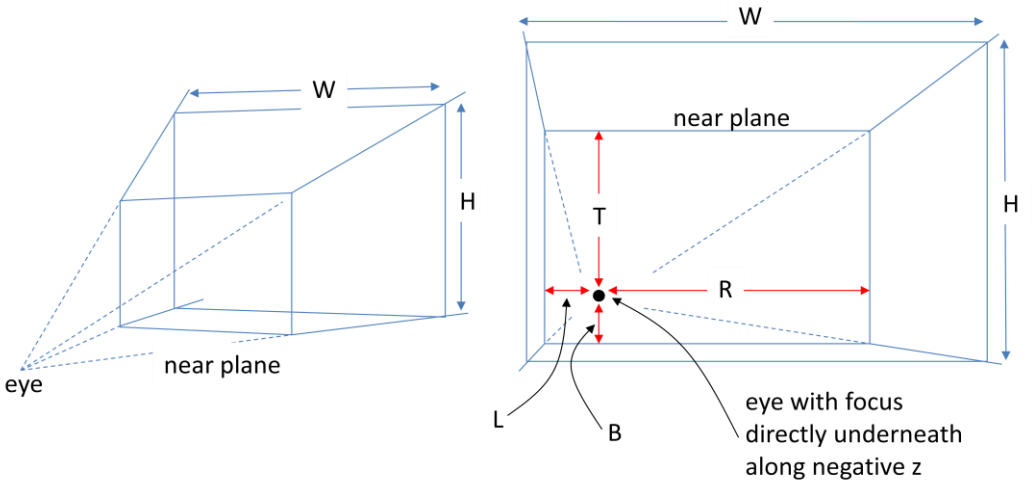
## Fourth stage – move the camera and frustum along with the glasses

All of the frusta we've used so far have been symmetrical and static, calculated from a single eye point. To make a head-tracked application we now need to re-calculate the frustum at every frame according to where the glasses have moved to since we last drew the scene. This changes its shape every time the glasses move left, right etc, so that most of the time the frustum will not be symmetrical any more. We also need to change the camera used to view the scene, so that it moves with the glasses too. The result will be a 'first person' view of the objects in front of the user, namely our monitor and cube. Because we made the monitor the same size as the graphics window, provided the calculations are correct the cube should draw at its real size too.

**Alter** the 'bInperson else if' in ViewSystem so that the 'eyepos' vector is set to the 'Pos' real-time vector and the focus is set to a place always directly in front of 'eyepos', but some fixed distance down the *negative* z axis (OpenGL axial system). Make sure the user can never get beyond the focus point– setting its z coordinate value on or a bit beyond the monitor should do it. The glasses' x, y, z coordinates are in the 0, 1, 2 elements respectively of vector 'Pos' – for example, Pos[0] will contain the x value. *However*, remember that the glasses is a tracker object described in the tracker axial system, so you will need to look at the running program to work out what order to put the components into the 'q_vec_set' calls and which ones have negative signs, because the 'look' matrix is in the OpenGL part of the pipeline. For example, to find the x value in the 'q_vec_set' call (second parameter – use the Intellisense for a hint), look at the OpenGL X axis in the scene and decide which tracker axis (X, Y, Z, same direction (positive) or opposite direction (negative)) it corresponds to. If you have to negate a coordinate you do -Pos[1] (for example), not Pos[-1], which is programmatic garbage.

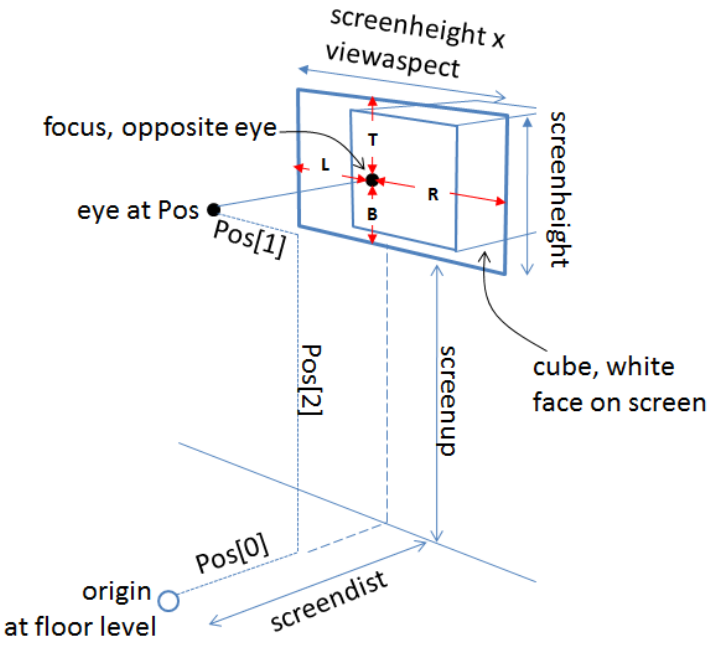
**Test** your program after this part – the monitor and cube will swim about and change size but if they stay in view most of the time you've probably done the transformation correctly.

We now have a symmetric frustum moving with the glasses, and you could probably relate what was seen to the glasses' movement by switching back and forth between the 'i' and 'p' presets. However, when we move around in the real world the objects do not swim about – we perceive them as fixed whilst we move. To do that we need a frustum where the drawing is fixed in space even though the eye moves. In terms of the diagram on the previous page this is like keeping the large $W \times H$ rectangle stationary even though the near plane (and therefore the eye) cranks around in front of it:




We tackle this in two parts, first finding the L, R, T, B values on the $W \times H$ drawing plane, then scaling them as we did in stage 3 to fit the near plane. At stage 3 we were able to calculate the L, R,

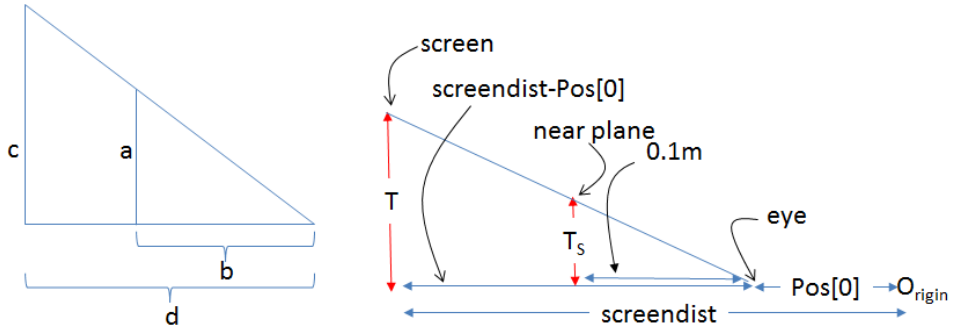
T, B values just once from the scene geometry, but now we're moving with the glasses we need to use the 'Pos' real-time vector and the actual *screen* dimensions, rather than the *scene* dimensions as we did before. This is because the scene now *is* the screen, seen from the user's viewpoint. If we take the middle of our screen to be at the same place as the monitor's black cross, and the origin to be on the physical floor, Pos and the other variables we've used so far fit together as follows:



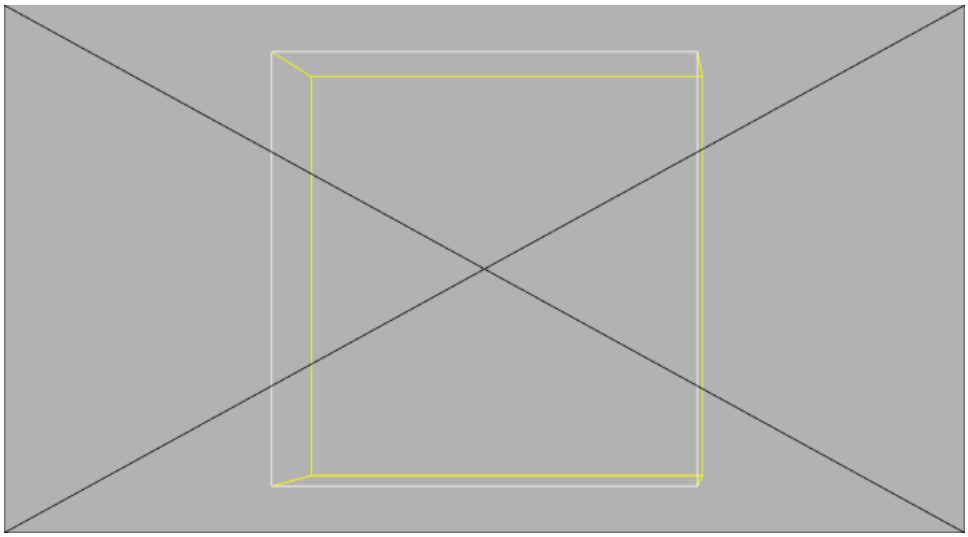

Because the origin is opposite the vertical midline of the screen, L (or 'unsignedleft' as it is called in the code) is given by $L = \frac{screenheight \times viewaspect}{2.0} - Pos[1]$ and, conversely, R is given by $R = \frac{screenheight \times viewaspect}{2.0} + Pos[1]$. Adding these two expressions together gives us the full width of the screen, as we can see it should from the diagram. The quantity B is obtained from $B = Pos[2] - screenup$ whilst T is given by $T = screenup + screenheight - Pos[2]$. These equations don't look symmetrical like those for L and R because the origin doesn't lie between in this direction, but adding together T and B still gives the screen height, as it should.

**Set up** the variables 'unsignedleft', 'unsignedright', 'unsignedbottom' and 'unsignedtop' according to the above equations. Put them into the AsymmetricFrustum with the correct signs – revise stage 3 if needed. Don't make any new variables and don't run the program yet because we're not quite done.

These equations calculate the quantities as if measured on the screen, whereas actually they should be scaled as if they were measured on the near plane. Similar triangles gives $\frac{a}{b} = \frac{c}{d}$ so by knowing the near plane is 0.1m from the eye we can calculate the scale factor:



For example, for T, the scaled distance $T_S$ is given by $T_S = S \times T = \left(\frac{0.1}{screendist - Pos[0]}\right) \times T$ or in other words $S = \frac{0.1}{screendist - Pos[0]}$. All the scale factors are the same so multiplying each L, R, B, and T parameter in AsymmetricFrustum by this variable will render the cube at its correct size, i.e. 'cubefacesize'. Its white face will be stationary but the yellow cube body beyond the screen will shear left, right, up and down as the user moves:

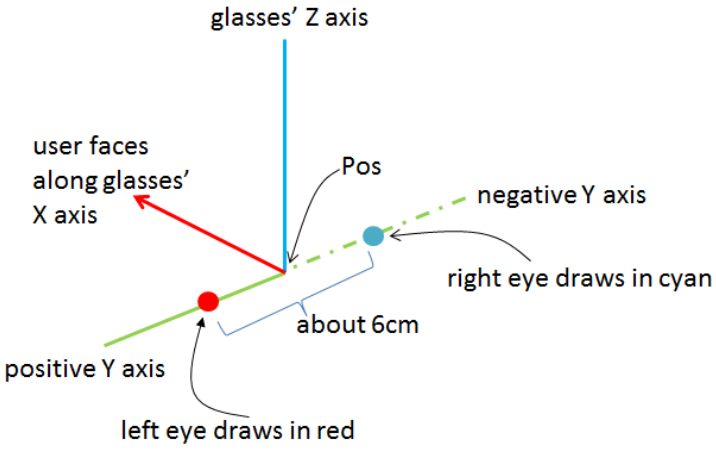


**Assessment by program demonstration recorded as a Canvas quiz [6 marks]:**

1. Camera looks from the glasses' position **[2]**
2. At a centred and stationary monitor/cube **[2]**
3. Cube size unaffected by changes in
    a. 'screenup' **[1]**
    b. 'screendist' **[1]**

## Fifth stage – stereo and the meaning of negative, zero and positive parallax

The final requirement is to take account of the user's two eyes. So far we have assumed the user is a cyclops, with just one eye sitting at Pos. Now we need to use the Y axis of the glasses object to find each eye's position in turn, and render the scene twice per pass from these two slightly different locations:



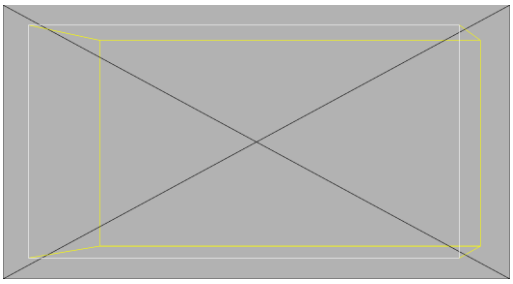

The Yaxis variable is a unit vector so to step along it by a small distance we just scale it, or in other words use a small scalar multiple of it. Expressed as vector algebra, Pos therefore becomes $\overrightarrow{Pos} \pm \delta \times \overrightarrow{Yaxis}$, where $\delta$, a scalar, is half the eyes' separation distance. You can decide using the above diagram which eye uses the 'plus' form of this equation and which uses the 'minus' form.

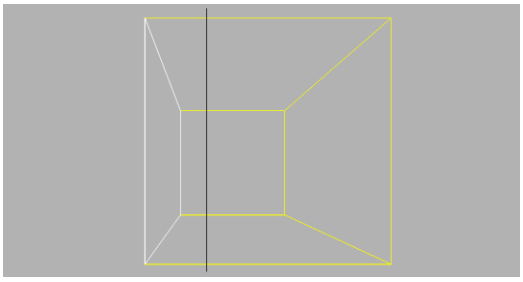
**Insert code** in the two empty { } at the top of ViewSystem to set 'Pos' to be the right eye's position and the left eye's position. By setting 'Pos' here, the rest of the code in this method continues to be used unchanged. Use the 'q_vec_???' routines (the Intellisense will give a useful hint – all of the 'quat' package routines are sensibly named) for clarity, and do not introduce any new variables. The 'Yaxis' variable is not used for anything else so you can safely overwrite this one with the scaled version you need. However, the 'Pos' variable is re-used in the second pass before being refreshed by the tracker update, so you will have to take this into account when computing the values for the left eye. The comment in the code contains a suggestion for how to think this through.

**Test** your program at this stage ('2' key). You should see the white face unchanged because at this stationary point there is zero parallax. The images for the left and right eyes overlay exactly and sum to white. The body of the cube separates into pink (left eye) and pale green (right eye), which are the colour components of the original yellow. This portion of the scene is behind the screen and renders in positive parallax, so-called because the separations move in the same direction as the corresponding eye. Refer to the diagram above and you can use this to check you chose the correct 'plus' and 'minus' form for the respective eye.

To render something in negative parallax, where the separations move in the opposite direction to the corresponding eye, requires the object to protrude out of the screen. We could simply move the cube towards the user but in doing so we would lose the stationary face and its diagnostic benefits. However, if we move the cube *and* make it the same aspect ratio as the screen, the edges parallel to OpenGL's Z axis will always intersect the cross of the monitor somewhere. Here on the screen they will be stationary points once again, with zero parallax.

**Scale** the cube to make a rectangular box whose width-wise and vertical dimensions have the same aspect ratio as the screen's width and height. Make the depth dimension the same as the vertical dimension. **Change** the box's translation so that it is centred when seen face-on but one-quarter out of the screen and three-quarters beyond the screen. It looks like this when viewed *without stereo* using the 'p' and 's' camera presets:





**Assessment by program demonstration recorded as a Canvas quiz [6 marks]:**

1. Positive- and negative-parallax separations ('2' key) **[1]** in a direction appropriate to each eye **[1]** along the $\pm$Y axis of the glasses **[1]** by an appropriate amount **[1]**
2. Appearance unaffected by changes in
    a. 'screenup' **[1]**
    b. 'screendist' **[1]**

## Feedback …

… is given verbally at each assessment demonstration, when you will also find out how many features were correct. The form of each question in the reporting quiz reflects the marking scheme in this document so a 'False' response selected by your assessor automatically records how you lost marks. You can view 'your' responses at the time and afterwards so use this feedback to improve your next deliverable if needed. Again, please note that you cannot revisit an assessment, so only ask for a stage to be marked if you have compared your features with the points in this document, and are sure you want to go ahead.

## Individual Canvas quiz – checks your understanding of the concepts [8 marks]

***Rules***: The quiz is done and marked via Canvas at the final lab on 1 November. In order to be given the access code you must have at least 6 so-called 'Promptly Points'. These are earned by demonstrating each numerical stage in the correspondingly numbered lab (2 points) or the week after (1 point). After that you still get the assessment points but no 'Promptly Points', and you have to demonstrate everything and in the prescribed order. The aim should be obvious – it's to get people working seriously straight away, rather than leaving everything to the deadline. Note that you can catch up to an extent because you can demonstrate more than once per lab, but there will be a queuing system in operation to ensure fairness to all, with strictly one assessment per queued request. Any obviously non-serious assessment attempt (sacrificing marks to earn the promptlies) will be disqualified and the lecturers' decision on this matter will be final. Please see the relevant Canvas quiz for the recording mechanism for 'Promptly Points'.

***Sample*** questions:

1. Axial transformation – convert the OpenGL point (3, 2, -1) to tracker coordinates ( , , )
2. Axial transformation – convert the Tracker point (1, 2, -1) to OpenGL coordinates ( , , )
3. Translations – an object is translated using a translateObject(2, 1, 0) then a translateObject(4, -2, 3). Write out the resulting single translateObject( , , ) equivalent.
4. Scaling – an object is scaled using a scaleObject(3, 1, 3) then a scaleObject(2, 1, 3). Write out the resulting single scaleObject( , , ) equivalent.
5. Similar triangles – a triangle abc has sides [a, 5.2, 9.1] and similar triangle αβγ has sides [2.0, 4.0, 7.0]. The value of a is *(insert <u>one</u> decimal number).*
6. Symmetric frustum – a SymmetricFrustum(55.0, 1.5, 3.0, 20.0) sets up a window whose height is *(state in <u>words</u>)* its width.
7. Asymmetric frustum – an AsymmetricFrustum(1.2, 2.2, 1.5, 3.0, 3.0, 20.0) sets up a window 3m from the user *(note there are <u>no</u> minus signs missing)* whose
    a. width is *(insert <u>one</u> decimal number)*
    b. and whose height is *(insert <u>one</u> decimal number).*

(Note that for clarity the pseudo-code calls do not include the first matrix parameter.)

You are very welcome to discuss tackling the sample quiz questions with staff and peers (live or on Canvas) as that is part of the understanding, and you don't have to wait until the final lab to do this. You are also allowed to use your program to help you answer, or check your answers for, the sample, but not during the test itself because this disadvantages people who are less sure of their programming skills. **Finally**, when taking the Canvas quiz for real in the final lab on 1 November, you **must** do it yourself and any collusion will be treated as an attempt to use unfair means. Scrap paper for rough working will be provided.