1. Vertex Performance (Spinning Tops)

a) This question begins by measuring the performance of the current **DrawModel()** subroutine, specifically the code between the first call to **GXBegin()** and the last call to **GXEnd()**. The original code renders the spinning top in three parts where each part has its own **GXBegin()** and **GXEnd()** call. The original code renders these sub parts in the following order;

1. Top (square based pyramid)
2. Middle (cube)
3. Bottom (square based pyramid)

Note: When timing this routine we observed that the results being output to the console by the PRINTPMC call, varied considerably per frame, especially the cycle count. For this reason we decided to average all of our readings across 80 frames to get a more accurate result.

Drawing Order	Primitives Used	Cycles	Instructions	Total Number of Vertices	Number of Vertices (Triangles)	Number of Vertices (Quads)	Number of Triangles / Quads	Vertex/ Triangle Ratio	Vertex/ Quad Ratio
(a) pyramid cube pyramid	TRIS + QUADS	1834	1411 (1488 every 4th render loop)	40	24	16	8 / 4	3:1	4:1
(b) pyramids cube	TRIS + QUADS	1747	1345 (1422 every 4th render loop)	40	24	16	8 / 4	3:1	4:1

b) From observing the above table, it can be seen that there has been a performance increase of **4.7%** by drawing both pyramids within the same **GXBegin()** and **GXEnd()** block. As well as noticing this performance increase, we have also noticed a skew in the instruction count every 4th **DrawModel()** call. Although at this stage we have no direct evidence as to why this might occur, we have found that the difference between instruction counts every 4th **DrawModel()** cycle was consistent across both code variations (77 in both instances). Our prediction for why this might be happening is that the 77 extra instructions every 4th frame are a result of ***context switching***[1] occurring on the GameCube, suggesting that another thread such as the NGC OS may be issuing 77 instructions.

[1] The process of saving an executing thread or process and transferring control to another thread or process. (docs.rinet.ru/NTServak/glossary.htm)

Drawing Order	Primitives Used	Cycles	Instructions	Total Number of Vertices	Number of Vertices (Triangles)	Number of Vertices (Quads)	Number of Triangles / Quads	Vertex/ Triangle Ratio	Vertex/ Quad Ratio
(c) pyramid pyramid cube	TRIANGLE STRIP + TRIANGLE FAN	915	653 (730 every 4th render loop)	22	12	10	8 / 4	3:2	5:2

c) For this part of the question we were asked to change the vertex format such that a triangle strip was used to form the middle cube, and triangle fans used to form the top and bottom square based pyramids. Using these vertex formats instead of the previously used "triangles" and "quads" has allowed us to improve the vertex to primitive ratios. We now use on average 1.5 vertices per triangle and 2.5 vertices per quad. It is also important to notice that we again see an additional 77 instructions every 4 draw calls, further supporting our theory that this is being caused by a background routine on another thread.

Here is the listing of code for the timed section of this program…

```
//RESET performance counters
RESETPMC
STARTMMCR1
STARTMMCR0

//Begin drawing in triangle fan mode
GXBegin(GX_TRIANGLEFAN, GX_VTXFMT0, 6);

//Start using vertices at offset +0 from the vertex buffer
for ( ivert = 0 ; ivert < 6 ; ++ivert )
{
	GXPosition1x8((u8)(ivert ));
	GXNormal1x8(iface);	// same normal for each vertex on a triangle
}
GXEnd(); //Break required to separate top and bottom sections of spinning top

//Begin drawing again in triangle fan mode
GXBegin(GX_TRIANGLEFAN, GX_VTXFMT0, 6);

//Start using vertices at offset +6 from the vertex buffer
for ( ivert = 6 ; ivert < 12 ; ++ivert )
{
	GXPosition1x8((u8)(ivert ));
	GXNormal1x8(iface);	// same normal for each vertex on a triangle
}
GXEnd();

//Begin drawing again in triangle strip mode
GXBegin(GX_TRIANGLESTRIP, GX_VTXFMT0, 10);

//Start using vertices at offset +12 from the vertex buffer
for ( ivert = 12 ; ivert < 22 ; ++ivert )
{
	GXPosition1x8((u8)(ivert ));
	GXNormal1x8(ivert);	// same normal for each vertex on a square
}
GXEnd();

//Stop timers and prints results
PAUSEMMCR0
PAUSEMMCR1
PRINTPMC
```

2. PowerPC Optimisation

This question involved investigating a question from a previous exercise done in class. This question simply used the shoulder buttons on the NCG controller to toggle the colour of two triangles drawn on screen. The left trigger toggles the colour of the triangle on the left and the right trigger toggles the colour of the triangle on the right.

a) *"By examining the compiler generated assembler, explain the difference in cycles between the colour transitions..."*

i) ...in the original C code

We decided that the best way to approach this was to show the execution paths of both scenarios (red →blue and blue →red).

Key

- = Branch instruction
  ☹ = Pipeline stall
  → = Control flow

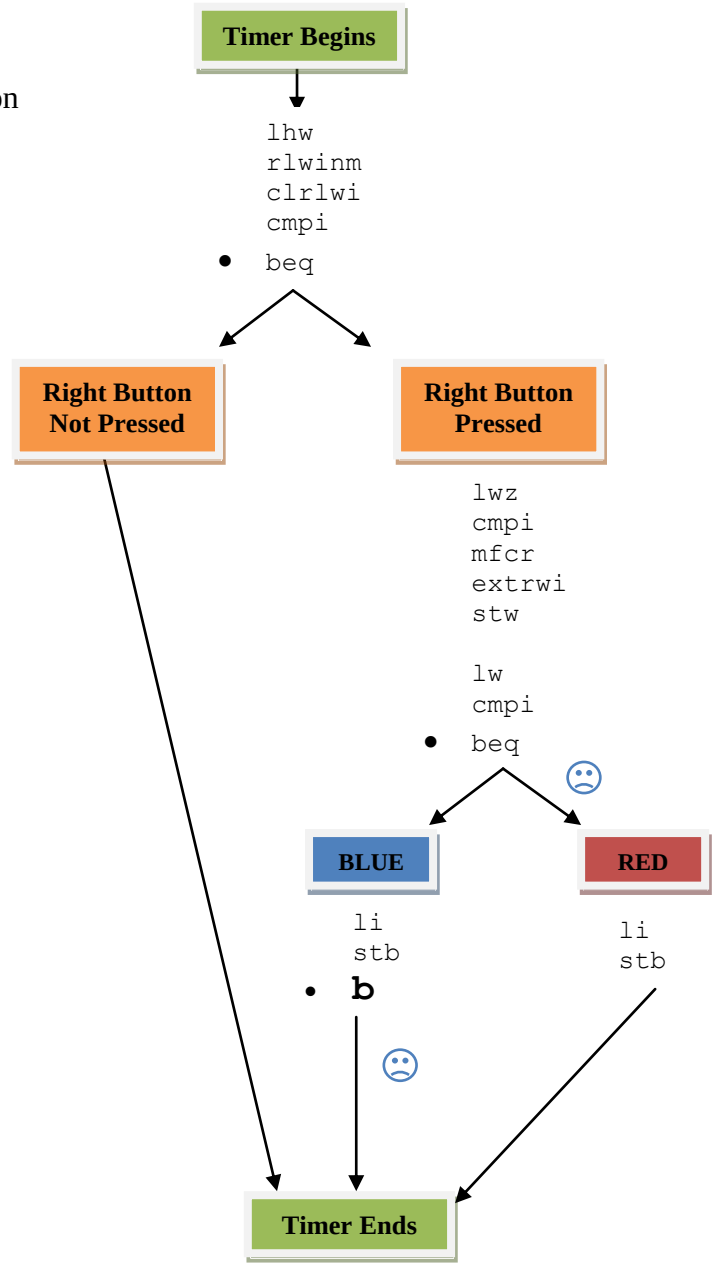


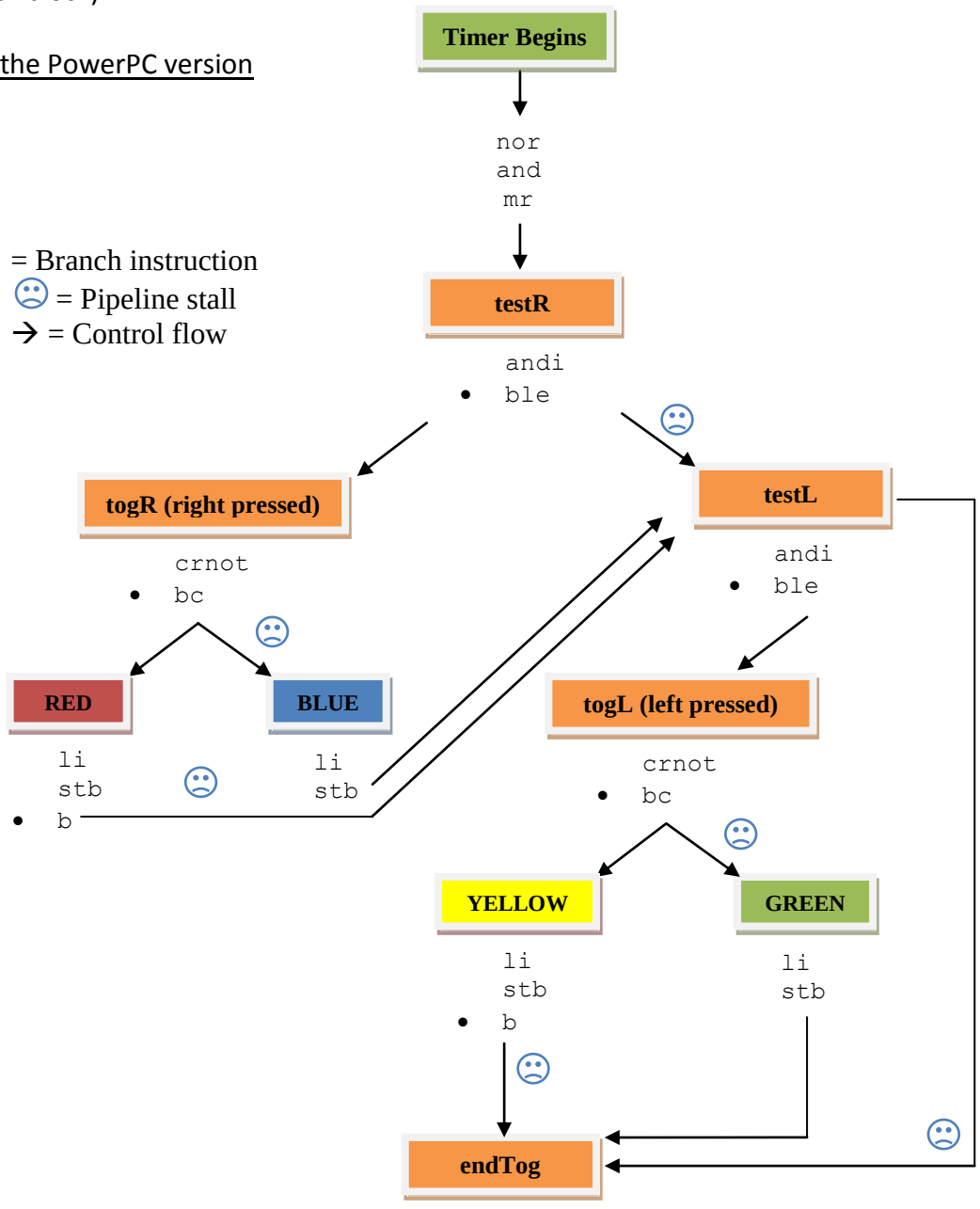
Diagram explained

At first we were suspicious that it could be branching instructions which were causing the difference in cycles between colours. Our reasoning for this was that any branch condition causes the instruction pointer to be changed, and in turn, flushes any pre-fetched instructions, causing a ***pipeline stall***. Although this is true we could not immediately hold branch instructions responsible for the cycle count difference, as each code flow path (both red and blue) contained one branching instruction/pipeline stall. However, we did observe that one **extra branching instruction** was being issued along the blue code path, to allow the code to jump past the else/red block. We did find it unusual that the compiler had chosen to produce assembly code in this manor, as we have always been instructed that the compiler should make the "if" route of an "if/else" selection, the most optimised. However with this in mind we could apply the same theory to the code toggling the left triangle between yellow and green (toggling to green is slower because the code to do so is in the "if" part of the "if/else" block).

ii) …in the PowerPC version

Key

- = Branch instruction
  ☹ = Pipeline stall
  → = Control flow



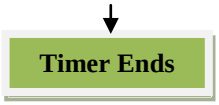


Diagram explained

When analysing the PowerPC version of the colour toggling code we realised that we could again apply the theory suggested above. This case was slightly different however, as the hand crafted PowerPC code implements the "else" part of the "if/else" as the least optimised route (as we had wrongly expected with the compiler produced version). For this reason the cycle counts results are inverted. With this approach, toggling to blue is now faster than toggling to red and the same with green and yellow for the left triangle. In effect this is a subtle use of branch hints. The original C version hints that the "else" part of the "if/else" selection will be taken most frequently (although as we are toggling this will never be the case) and the opposite for the PowerPC version.

b) Further optimising the PowerPC code

Modification 1

The first modification we made to the assembly code was to remove the double use of register r19 as we were suspicious that reading and writing to a register in the same instruction may possibly decrease performance. Although this change did not bring about any apparent performance increase, it did add clarity to the procedure and separated out variables being used.

Original

```
nor         %r19, %r18, %r18   # downs = ~lastbutton
and         %r19, %r19, %r17   # downs = downs & pad[0].button
mr          %r18, %r17          # lastButton = pad[0].button
```

Modified

```
nor         %r22, %r18, %r18    # r22 = ~lastbutton
and         %r19, %r22, %r17    # downs = ~lastbutton &pad[0].button
mr          %r18, %r17           # lastButton = pad[0].button
```

Modification 2

Our next observation was that for every render cycle we were repeatedly loading the colour index values into register 0 using the ***li*** (load immediate value) command. One benefit of the PowerPC architecture is that we have 32x32bit registers at our disposal, allowing us to load each colour index into a separate register **outside** of the render loop.

Outside the while loop

```
//Load colour indices into registers
asm("li %r23,0"); // yellow
asm("li %r24,1"); // green
asm("li %r25,2"); // red
asm("li %r26,3"); // blue

#other initialisation code

while(!(pad[0].button & PAD_BUTTON_START)) //etc
```

Inside the while loop

The li instructions previously inside the while loop have become redundant and we can now directly copy in each colour index by selecting the appropriate register (from23→26)

```
red:    stb  %r25, 0x0(%r21)   # colorIndex[1] = RED
blue:   stb  %r26, 0x0(%r21)   # colorIndex[1] = BLUE
yellow: stb  %r23, 0x0(%r20)  # colorIndex[0] = YELLOW
green:  stb   %r24, 0x0(%r20)  # colorIndex[0] = GREEN
```

Modification 3

We next noticed a possible pipeline stall on the third line of the de-bounce code. The second line does in fact have a dependency on the first line as it requires the use of register 22 which is set on line 1. However, the third line of the de-bounce code did not need to immediately follow the previous line on which it had a dependency (access to register 17). As the result of line 3 is stored in register 18, and register 18 is not accessed again until the next iteration of the loop, we moved line 3 to the very end of the procedure, as to remove the dependencies between lines 2 and 3.

Original

```
nor         %r22, %r18, %r18    # r22 = ~lastbutton
and         %r19, %r22, %r17    # downs = ~lastbutton &pad[0].button
mr          %r18, %r17           # lastButton = pad[0].button
```

Modified

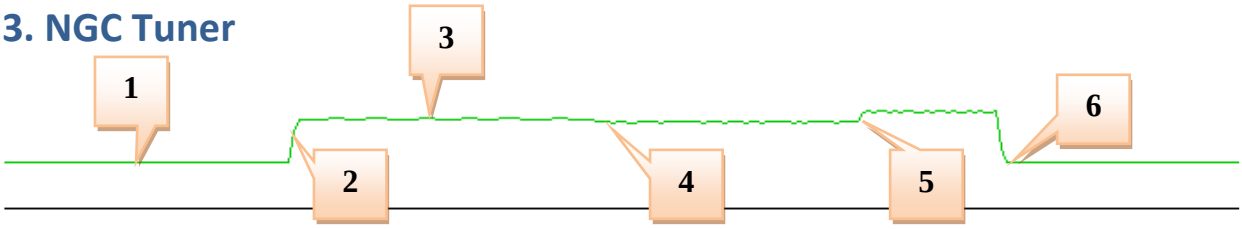
```
nor         %r22, %r18, %r18    # r22 = ~lastbutton
and         %r19, %r22, %r17    # downs = ~lastbutton &pad[0].button

#rest of loop code

endTog:     mr %r18, %r17           # lastButton = pad[0].button
```

Results

Overall the modifications we have made in this question have slightly improved the performance of the original PowerPC Code. Modifications 1 & 3 combined have reduced the number of quiet cycles by 4. The number of quiet instructions however remains the same (proving that re-arranging the order of execution has had a significant effect on cycles used). Modification 2 has removed one instruction per colour toggle, which in turn has reduced the cycle count by one. Here is a final table showing the results of our efforts.

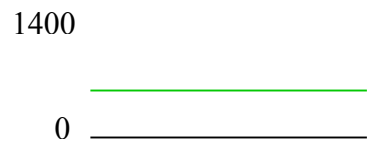
Version	*Action*	*Cycles*	*Instructions*
Original PowerPC	Quiet	14	8
	LTrigger	12 (blue) / 17 (red)	11
	RTrigger	18 (green) / 21 (yellow)	11
Optimised PowerPC	Quiet	10	8
	LTrigger	11 (blue) / 16 (red)	10

	RTrigger	17 (green) / 20 (yellow)	10
	Totals:	-4 (Quiet) / -1 (Toggling)	-0 (Quiet) / -1(Toggling)



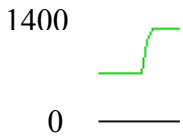
a) This question focuses on the use of the ***NGC Tuner*** software to profile the code contained in "tuner-cubes.c". For the first part of this section we were asked to form a trace, measuring instruction cache misses as we toggled through "scenes" in the GameCube application. Access to the GameCube's Instructions Cache (ICU) requires 1 cycle. However, if the desired instruction does not reside in the ICU it must first be fetched from system memory at a cost of 32 cycles. Therefore, as a programmer it is always useful to be aware of what is in the cache and when, in order to minimise cache misses, although obviously due to the limited size of the GameCube's cache memory, an IC_MISS cannot always be avoided.

Part 1



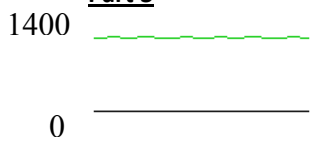
This is the first segment of our trace which represents the time when the system is idle and drawing no objects on the scene. We can see that we have a constant rate of 527 instruction cache misses per frame. We would expect this to be constant if the state of the game is not changing. We would not be expecting zero cache misses as the NGC's instruction cache is only 32KB and would not be able to store the program code in its entirety (the debug build comes in at 1.36mb!).

Part 2



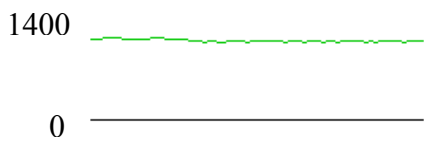
Part 2 sees the instruction cache miss count rise from 527 to 1110 over the space of two frames. This is the stage in the game where the z trigger is pressed and the scene renders a collection of cylinders. One of the reasons for the increase is that now we are actually calling some draw methods, the number of branching instructions will increase, meaning more data is being pushed in and cleared from the instruction cache, hence increasing the miss rate.

Part 3



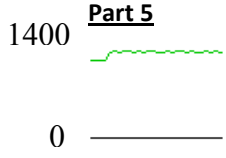
At part 3 we can see that the instruction cache miss rate is fluctuating continuously between 1106 and 1124. Our only prediction as to why this might be happening is that the branch prediction unit may be influencing the instruction pre-fetching which is alternating every few frames.

Part 4



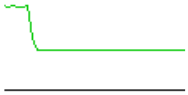
At part four we see the transition between cylinders and cubes take place. There is a slight fall of and the average number of instruction cache misses falls to ~1072. This may be due to the fact that it is less computationally expensive to generate vertices for cubes than it is to generate vertices for cylinders and there is no trigonometry involved in the computation (the ***kernel_sinf*** and ***kernel_cosf*** methods are not invoked during this period). Each call to a sin or cosine function is another branch and another potential instruction cache flush.

Part 5



This is the stage of the application where we switched to the final scene where we draw a collection of tori to the screen. This is probably the most computationally expensive primitive to generate as it again uses trigonometry and parametric equations to define the vertices and normals for the 3D primitive.

Part 6



At this stage we have cycled through the three scenes and return to idle mode where no primitive are drawn to the scene. We are now in the same state as in part 1.

Scene	*Time for rotation (seconds)*
1 (Cylinders)	6
2 (Cubes)	6
3 (Tori)	18

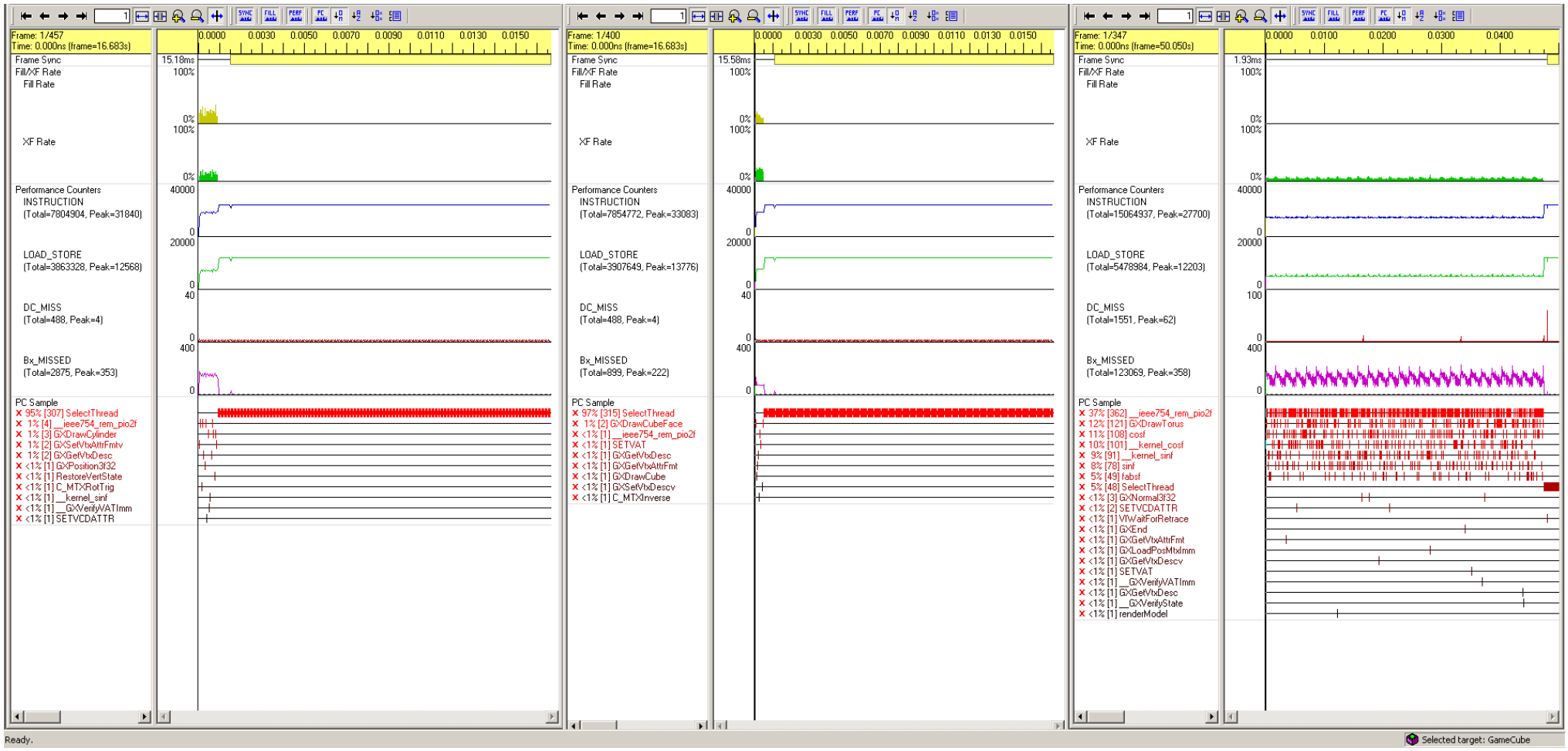
b) As can be seen by the timing results, both the cylinders and the cubes take 6 seconds to perform a 360° rotation around the Y-axis. However, the tori take 18 seconds, 3 times slower than both the cylinders and cubes. Before we observed the NGC Tuner captures, we hypothesised as to why this might be the case. We concluded that the main reason behind this time increase is that the torus is a much more complicated shape in comparison to a cylinder or cube. With a torus, there is a larger amount of polygons to render in comparison to a very simple shape such as a cube increasing the number of vertices and surface normals required. Having more vertices and normals to generate and manipulate has knock on effects at different stages of the rendering pipeline, most affected are the transform and rasterization stages.

We then went onto capturing the rotation process in the NGC Tuner. We decided to capture the following 4 variables:

- Instructions called per frame
- Loading from memory and storing to memory
- Data cache misses
- Conditional branch misses

The graphs produced by the tuner can be seen on the following page.

Profile Results



Cylinders
Frame time: 16.683ms

Cubes
Frame time: 16.683ms

Tori
Frame time: 50.050ms

By analysing the 3 captures, the most noticeable difference is the average render time per frame. Both the cylinders and cubes have an average render time of 16.683ms whereas the tori have an average render time of 50.050ms. When observing the captures in further detail it is easy to see why the tori take so much longer to render. As can be seen in the "BC_MISSED" column a high amount of branching is occurring every render cycle when rendering a torus, whereas much fewer branches are taken when rendering a cylinder or cube. The extent of the branching when rendering a torus can be seen clearly when observing the PC Sample and knowing what functions are called per frame. When rendering a torus it can be seen that floating point operations occur 37% of the time, per frame. It can also be seen that both sin and cosine calculations occur 38% of the time. These statistics further prove the theories we suggested.

c) For the final part of question 3 we were asked to replace the scene building code with an alternative code listing which allowed us to toggle between 2 different scene building approaches at runtime. We made the two approaches render the scene in different colours as to make identifying each approach easier. He two code approaches given are a nice example of how generating the same output while using a different technique can have a substantial impact on performance. More specifically this demonstration shows how when using loops, keeping branching and loop variants down to a minimum can improve performance.

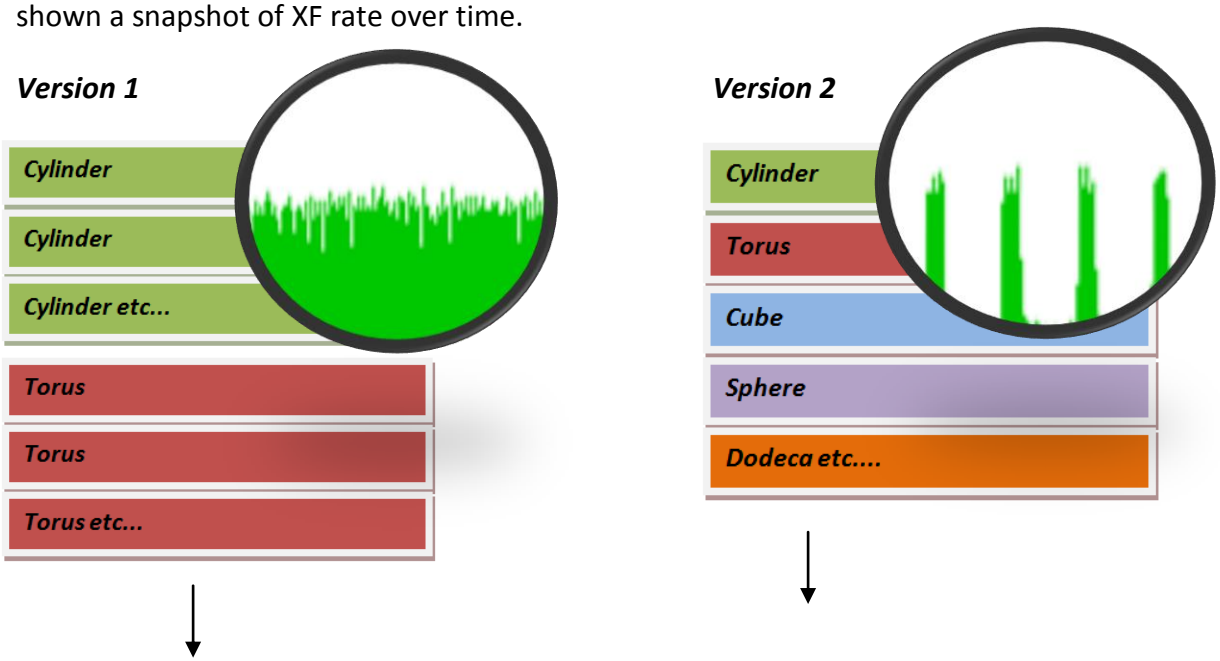
Here is the scene building listing given to us as part of this question.

```
/******************** SCENE BUILDING LOOPS  ****************************/

//draw 5 slice 5 X 5 cube of primitives (version 1)
if(scene == 1)
        {
        for(Trans.x = 6.0; Trans.x >= -6.0 ; Trans.x = Trans.x - 3.0)
        {
                eTrans.x = efac * Trans.x;
                for(Trans.y = 6.0; Trans.y >= -6.0; Trans.y = Trans.y - 3.0)
                {
                        eTrans.y = efac * Trans.y;
                        for(Trans.z = 6.0; Trans.z >= -6.0; Trans.z = Trans.z - 3.0)
                        {
                                eTrans.z = efac * Trans.z;
                                renderModel(slice[sliceNum], vr, eTrans, Scalar, mRdegs, Colours[6]);
                        }
                }
                sliceNum++;
        }
}

//draw 5 slice 5 X 5 cube of primitives (version 2)
if(scene == 2)
{
        for(Trans.z = 6.0; Trans.z >= -6.0 ; Trans.z = Trans.z - 3.0)
        {
                eTrans.z = efac * Trans.z;
                for(Trans.y = 6.0; Trans.y >= -6.0; Trans.y = Trans.y - 3.0)
                {
                        eTrans.y = efac * Trans.y;
                        for(Trans.x = 6.0; Trans.x >= -6.0; Trans.x = Trans.x - 3.0)
                        {
                                eTrans.x = efac * Trans.x;
                                renderModel(slice[sliceNum], vr, eTrans, Scalar, mRdegs, Colours[6]);
                                sliceNum++;
                                if(sliceNum == 5) sliceNum = 0;
                        }
                }
        }
```

sliceNum selects the 3d model which is drawn in renderModel. Therefore each time we change the value of sliceNum we are telling the system to switch over to draw a new object. In this version of the scene building loop, we change this number **5** times.

In this version of the code we not only add the added work of regulating the value of sliceNum, but also have to change the rendered model every step of the way.

Here is a small diagram showing how often the model being drawn is changed. It is worth noting that overall the same number of each model is drawn, each in the same position on screen, hence both methods producing an identical result. Against each diagram I have also shown a snapshot of XF rate over time.



The XF rate shows the activity of the GameCube's transform engine which is used to transform vertices from different spaces (model →world→view→projection). If we were to draw 10 cubes in a row for example, we could retain data from the first cube draw call and re-use some of the computations to draw the next 9 cubes. However, when switching between models rapidly (as demonstrated in version 2 of this application) we cannot retain or re-use data between draw calls and we are more likely to thrash the cache and other memory units.

Instructions / IC_MISS

Although this question suggests that we should comment on the IC_MISS data from the tests, we can see below that in both cases, the IC_MISS values seem very low and sparse. We are not sure why this is the case and can only guess that it is due to the granularity of the trace.