**GMAPS**

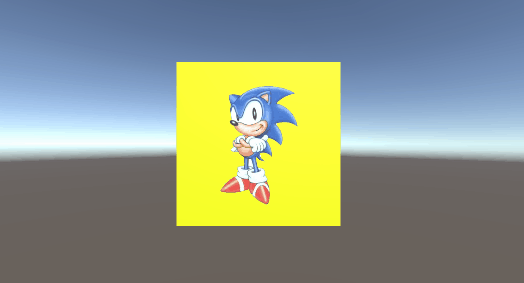
[A picture containing text, table, vector graphics

Description automatically generated](https://soundcloud.com/user-879892202)

**Matrices**

**Worksheet**

**To see the additional comments and resources, make sure you select All Markup in the Review/Tracking pane**



Important note

**PART 1: HMatrix2D**

**Question 1**

Unity has a **Matrix4x4** class, but just like for vectors, we’ll create our own class called (surprise!) **HMatrix2D**.

Download the file **HMatrix2D.txt** from the **Matrices** section on LMS:

Graphical user interface, text, application

Description automatically generated

This is a template for the **HMatrix2D** class you will create.

1. Open your **GMAPS\_Oct\_2023\_Worksheets\_STUDENT** Unity project, then add a new script called **HMatrix2D.cs** inside the **Math** folder.

This is a matrix for 2D coordinate space, but because of the h value, we must declare a 3x3 array to store the matrix values.

Do that now:



1. There are THREE constructor functions in the template code.

A screenshot of a computer

Description automatically generated with medium confidence

This means we have three ways to initialise a HMatrix2D object when we instantiate one.

Remember that if we multiply a matrix A with the **identity** matrix **In**, then the result of that multiplication is just **A**, i.e. no change.

So, we would like the first constructor to initialise the matrix to the identity matrix.

But we might want to set an HMatrix2D object to identity at any time, e.g. if we have to reset it for some reason. In this case, we should create a function to do this, which we can run any time.

Complete this code:

Timeline

Description automatically generated with medium confidence 1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

Create another C# script called **TestMatrix.cs**.

Text

Description automatically generated

Create a **new** **empty game object** in the scene called TestMatrix, add TestMatrix.cs to it, and run the project. You should see this output:

Graphical user interface, application

Description automatically generated

1. The code in **SetIdentity()** is rather verbose (long-winded).

It can be simplified using what is called the **ternary operator**.

Comment out (don’t delete) the original code, and replace it with the ternary operator version.

Chart

Description automatically generated

Text

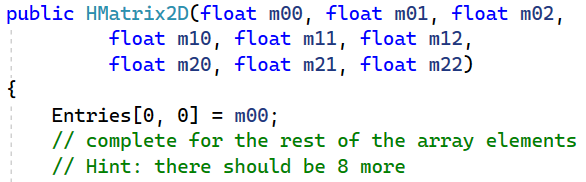
Description automatically generated

1. Now you must implement the other two constructors.

Chart, timeline

Description automatically generated

and …



Text

Description automatically generated

So far, you should understand what the HMatrix2D class is for, and how to create and initialise an HMatrix2D object.

This will be used to perform matrix math, rather than Unity’s built-in Matrix4x4 class.

**Question 2**

The next thing you should do is implement functions for the basic arithmetical operations for HMatrix2D: **add**, **subtract**, and **multiplication** **by a scalar** value.

1. Implement these arithmetical functions now.

Refer to your lecture slides and complete this code:

A screenshot of a computer

Description automatically generated with medium confidence

If you need a refresher on **operator overloading**, look here.

You also need to check if two HMatrix2D objects are **equal or not**.

1. Now, you must implement checking for equality.

Complete this code:

Chart

Description automatically generated

Inside the **if statement**, you need to check if any two corresponding elements are *not* equal.

If not equal, then the HMatrix3D objects aren’t equal, so return false.

If after looping across all the elements none of them cause a false return, then return true (the HMatrix2D objects are equal).



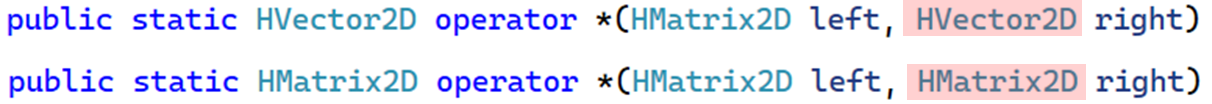
For the **!=** operator, simply reverse where the function returns true and false.

Now you have to think about the more complex multiplication of:

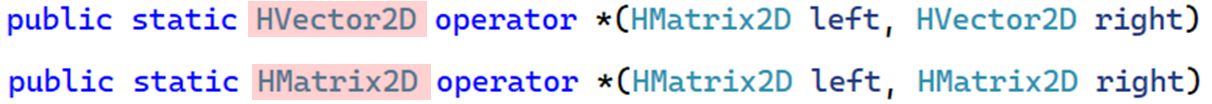
* matrix and vector
* matrix and matrix

In fact, the multiplication of a matrix with a vector is performed the same way as for two matrices, since a vector is really just a matrix.

But because our code defines HMatrix2D to represent a matrix, and HVector2D to represent a vector, we need a separate function for multiplying a matrix and a vector, as shown below:



Also, a matrix-by-matrix multiplication returns another matrix, but a matrix-by-vector multiplication returns a *vector*:



So, we need a \* function with arguments for HMatrix2D and HVector2D objects, and another function with two HMatrix2D arguments.

This is why we must overload the \* operator.

1. Implement the overloaded \* operator function for **HMatrix2D \* HVector2D**.

Chart

Description automatically generated with medium confidence

This multiplies left (a HMatrix2D object) and right (a HVector2D object), and returns a new HVector2D object.

Remember that HVector2D has three values: **x**, **y** and **h**.

So why does the function above only calculate the top *two* HVector2D values (for x and y)—what about h?

Looking at the HVector2D constructor, the answer is clear: you’ll see that it takes two arguments, **x** and **y**, and that the third value for **h** is automatically set to 1.

A picture containing text

Description automatically generated

Remember that HVector2D is a 2D vector, with only x and y values. But it also has an extra h value. The h value is always 1, so we can just set it automatically in the constructor.

1. Implement the overloaded \* operator function for **HMatrix2D \* HMatrix2D**.

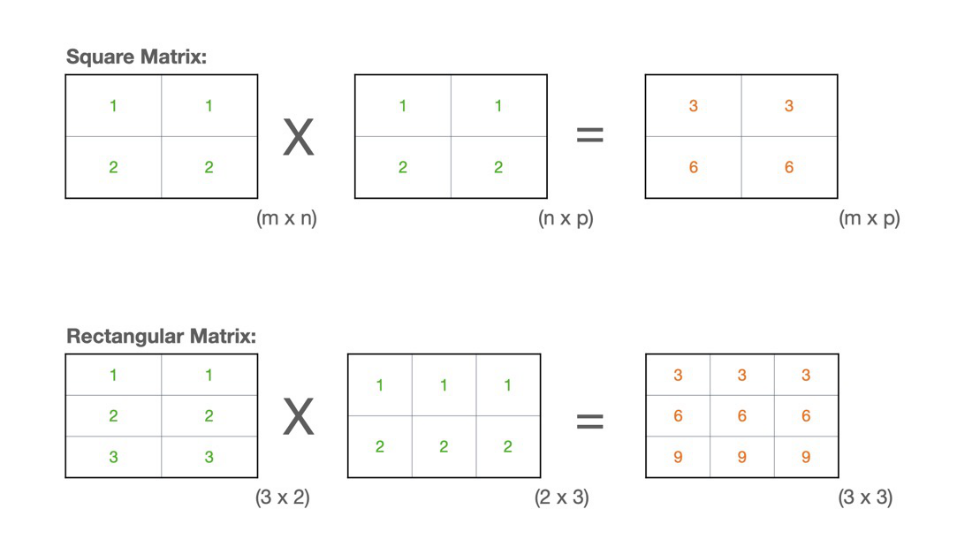
The first two elements in the array in the HMatrix2D template are given. You must complete the rest of the entries.

The code in the template simply hardcodes all the rows and column indexes, and then multiplies these together as shown in your lecture slides.

This is ok for small matrices, but 4x4 matrices and larger would need a more sophisticated approach.

If you search online, you might end up with more complex code that you won’t properly understand, and which isn’t *really* necessary for a small matrix.

*But* you can take a look at this link: <https://www.geeksforgeeks.org/c-program-multiply-two-matrices/>



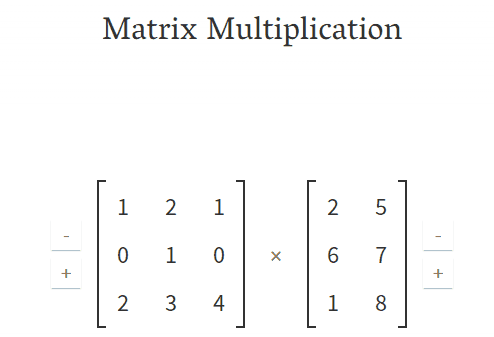
The algorithm can multiply together two square or rectangular matrices of any size.

If you’re able show that you understand the code when you submit your project (properly commenting every line, and with a high-level description of the algorithm in a comment above the function), then you can choose to use this more general approach instead.

1. In **TestMatrix**, add a function called **Question2()**.

Go to this URL:

<http://matrixmultiplication.xyz/>



Try various combinations of 3x3 \* 3x3 and 3x3 \* 3x1 matrices to multiply. You can step through to see the matrix multiplication in action.

Declare three HMatrix2D objects, **mat1**, **mat2**, and **resultMat**. Also declare one HVector2D object, **vec1**.

For each multiplication that you try at the website above, test that the result is the same when you run Question2() in Start().

Check that both overloaded \* operator functions work in your code.

Text

Description automatically generated

**PART 2: Application**

**Question 3**

Now you’ll implement **SetRotationMatrix** and **SetTranslationMatrix**.

1. Complete this code (refer to your lecture slides for 2D rotation matrices):

A picture containing chart

Description automatically generated

1. Complete this code (refer to your lecture slides for 2D translation matrices):

A picture containing diagram

Description automatically generated

Now you have most of HMatrix2D finished!

Before applying a matrix to transform a game object, we need to access the game object’s *vertices*.

A cartoon character on a yellow background

Description automatically generated

Look at **MATRICES\_worksheet/Sonic/MeshManager.cs**.

This uses Unity’s **MeshFilter** component to access a game object’s **sharedMesh** vertices and store these in a Vector3 array called **vertices**.

Here’s Unity’s documentation for MeshFilter:

* <https://docs.unity3d.com/Manual/class-MeshFilter.html>

Watch these Unity videos about object meshes:

* <https://www.youtube.com/watch?v=cKJ6dQfgcAU>
* <https://www.youtube.com/watch?v=m3pblzmvFpE> (only need up to 1:30)

Read this Unity forum question:

* <https://forum.unity.com/threads/mesh-vs-shared-mesh-in-procedural-generation.583957/>

**Question 4**

1. Answer the questions below. Make sure you read the code carefully. Expand the answer text box if necessary.
2. Explain *in your own words* what MeshManager’s Awake() function does.

The MeshManager will get the **MeshFilter component** and will use that component to set the values for the originalMesh, clonedMesh, vertices and triangle using the Mesh given by the **sharedMesh** from The **MeshFilter.**

1. Explain what problem will occur if you don’t use a *cloned* mesh in this code, and how to fix it.

Look here for a hint:

<https://www.kodeco.com/3169311-runtime-mesh-manipulation-with-unity/>

It will affect the original mesh. Once changes has been done to the original mesh, it is hard to revert the changes as the changes are permanent.

Text

Description automatically generated

1. Create a new script called **TransformMesh.cs** and add it to the Sonic prefab. Then drag an instance of the Sonic prefab to the scene. This should already have a **MeshManager** script attached (if not, add it).

Reset Sonic’s position to the origin.

Add these variables to TransformMesh:

A close-up of a white background

Description automatically generated

What each variable represents is discussed below.

|  |  |
| --- | --- |
| vertices | These are the copies of the vertices of the sprite mesh’s vertices which are copied and stored by MeshManager. These are used to transform (translate, rotate and scale) the sprite, otherwise the sprite mesh will have the transformation permanently applied. |
| transformMatrix | This is the matrix object, which is of type HMatrix2D (this is the matrix class you defined earlier in this worksheet). You will use the SetTranslationMatrix and SetRotationMatrix methods of this class to set transformMatrix to translate or rotate the sprite. |
| meshManager | This is an instance of MeshManager, which is used to obtain a copy of the sprite’s vertices. |
| pos | This is the current position of the sprite. When rotating or scaling, the sprite must be translated to the origin, then rotated or scaled, then translated back to its original position. See the diagram below. |

Look at the diagram below. To rotate or scale a game object, we must (a) first move it to the origin, (b) perform the rotate or scale transformation, and then (c) move it back again.

A picture containing shape

Description automatically generated

The matrices declared in (b) above represent these different transformation steps.

* **toOriginMatrix**

The translation matrix for translating a vertex position in step (a).

* **toOriginMatrix**

The translation matrix for translating a vertex position in step (c).

* **rotateMatrix**

The rotation matrix for step (b).

* **transformMatrix**

The final transformation matrix, which concatenates all the matrices above (we’ll look at this later), e.g., for rotation, we perform this matrix multiplication:



This lets us perform all the separate steps as a single step! In other words, multiple transformations can be concatenated into a single *composite* transformation. This is why we use *homogeneous* coordinates.

Remember that the matrix multiplication order is right-to-left.

* **pos** is the current position of the game object stored as a HVector2D object.

Text

Description automatically generated

**Question 5**

Next, we’ll apply the various transformations to the Sonic sprite mesh.

Complete the steps below. Make sure you read the code carefully.

1. Add the code below to **TransformMesh**:

A computer code with black text

Description automatically generated with medium confidence

**MeshManager** has been added as a component script to the Sonic sprite game object, so we obtain a reference to it, which is stored in **meshManager**.

Then, we store the current position of the Sonic sprite in **pos**.

If we run the scene, nothing happens! This is because we don’t have any methods for transforming the sprite.

We’ll add three methods:

* **Translate**
* **Rotate**
* **Transform**

1. Write the **Translate** method. It will look like this:

A screenshot of a computer code

Description automatically generated

You’ll get a compilation error, because you haven’t written the **Transform** function yet.

1. Write the **Transform** method. It will look like this:

A screenshot of a computer program

Description automatically generated

Now you have the Translate function and Transform function.

1. Modify the **Start** method so that the Sonic sprite is moved by (1, 1).

Hint: You need to call the **Translate** method. The sprite might move outside the game window, so view the transformation in the Scene view.

A screenshot of a video game

Description automatically generated

You have moved the sprite using a direct matrix multiplication! You didn’t use any built-in Unity API function calls, such as Transform.Translate().

Give yourself a high-five 😊

1. Now try to implement and use the **Rotate** function.

This is more tricky, and you’ll have to refer to the lecture slides.

Remember that we will use these matrices:

* **toOriginMatrix**
* **toOriginMatrix**
* **rotate Matrix**

These are multiplied together *in reverse order*, and stored in **transformMatrix**.

Some clues are given in the code below.

A screenshot of a computer program

Description automatically generated

Now you should be able to call the Rotate method to rotate the Sonic sprite mesh:  
  
A cartoon character on a yellow diamond

Description automatically generated

Again, you’ve used only direct matrix multiplication, with no Unity API calls. Another high-five! 😊

It’s left as an extra exercise for you to implement scaling.

**PART 3: Reflection**

**Question 6**

1. What was the most difficult part of this worksheet. Explain why you found it difficult, and how you managed to overcome this (if at all).

*You might like to reference specific lecture slides and lines of code, give some URLs and diagrams as references, etc.*

The hardest part of the worksheet is probably trying to understand what the worksheet requires me to do. I find the clues given are too vague and the function names does not really explain what I am supposed to do.

A picture containing chart

Description automatically generated

One of the question I am referring too.

I would have to sit down there trying to figure out what is it I am supposed to do because the clues given does not help. It is only when I confront to the teacher do I know what I am supposed to do. Since the worksheet depends on one’s implementation, the code written to function like **SetRotationMatrix** can have different implementation. So having the code as guidelines just does not help since I must figure out the implementation. Other than that, the topic taught were quite easy to understand and fun to learn.

1. Choose a matrix topic that are you still unsure about. Explain that topic, and the steps you will take to clarify your doubts. You will be graded on how well you analyse your problem with understanding that topic.

If you are confident about all the topics, select what you consider to be the most difficult topic and explain ONE more advanced concept related to the topic. You will be graded on the complexity of the concept, and how well you explain it.

*You might like to reference specific lecture slides and lines of code, give some URLs and diagrams as references, etc.*

I find it hard to understand the concept behind inverse matrix. Inverse matrix is a matrix where when multiplied by the matrix derived, would get an identity matrix.

1. Write a reflection about this worksheet. You may like to include how effective it is to motivate and help you learn, how well it relates to games development, etc.

(Min 150 words)