PHP Piscine

Day 06

Staff 42 piscine@42.fr

Summary:

This document is the day06's subject for the PHP Piscine.

Contents

Chapter 1

Foreword

Nothing for now.

Chapter 2

General Instructions

- Only this page will serve as reference; do not trust rumors.
- Watch out! This document could potentially change up to an hour before submission.
- Only the work submitted on the repository will be accounted for during peer-2-peer correction.
- As when you did C Piscine, your exercises will be corrected by your peers AND/OR by Moulinette.
- Moulinette is very meticulous and strict in its evaluation of your work. It is entirely automated and there is no way to negotiate with it. So if you want to avoid bad surprises, be as thorough as possible.
- Using a forbidden function is considered cheating. Cheaters get **-42**, and this grade is non-negotiable.
- These exercises are carefully laid out by order of difficulty - from easiest to hardest. We **will not** take into account a successfully completed harder exercise if an easier one is not perfectly functional.
- You cannot leave any additional file in your repository than those specified in the subject.
- Got a question? Ask your peer on the right. Otherwise, try your peer on the left.
- Your reference guide is called **Google / the Internet / http://www.php.net / ....**
- Think of discussing on the Forum. The solution to your problem is probably there already. Otherwise you will start the conversation.
- Examine the examples thoroughly. They could very well call for details that are not explicitly mentioned in the subject …
- By Odin, by Thor ! Use your brain !!!

Chapter 3

Introduction

Today your long and exciting journey into object-oriented programming, [OOP] begins in particular with PHP. During the past week, you have learned the basics of syntax and semantic of this language and we therefore assume that you are able to write code that can be used in PHP.

The main purpose of an object layer of a language has for main usefulness to facilitate the semantic splitting of your code and it is on this aspect in particular that today's exercises will focus. The PHP object syntax is very simple, the exercises will not follow the traditional Piscine exercise splitting, namely an exercise == a notion in particular. On the contrary, the following exercises will guide you through the realization of a small and fun program, and it will be up to you to deduct from the information we'll give you what notions are useful to complete the exercise. Yes, you'll have to think and compare your ideas.

For those of you with experience in OOP, you probably noticed that today's videos cover only a part of the OOP in PHP, the modular programming aspect to be precise. Of course, the rest of the concepts will be addressed tomorrow and the day after tomorrow. With regards to today's exercises, you must only use what has been covered up to this day. No exceptions, inheritance, traits, and other interfaces. This will be for tomorrow and the day after tomorrow.

During this Piscine, and in projects to come, you will have your batch of websites and other web applications to create. So why not take advantage of this day to do something else, 3D for example? You have done a Raytracer not so long ago, now we are going to do Rasterisation.

Several important things to note before you begin. Firstly, this day of Piscine will only be corrected by peer-correction. Therefore no need to be stressed over the Moulinette. The example outputs don't need to be pixel or space accurate, even if the exercises encourage a certain formatting that we invite you to observe. What we are interested in is the result and the structure of your code.

Secondly an amount of research is required on your part to complete today's exercises. Even if the technical concepts related to PHP are described in the e-learning section of this subject, it is your responsability to research both the vocabulary and concepts related to 3D.

Thirdly, contrary to what some statements may imply, the mathematical concepts required to achieve this Piscine day are minimal. If you lose yourself in mathematics, you are doing something wrong. We do not ask you to know how to build a projection matrix in front of an amphitheatre with explanation of all the details of transformation, accompagnied by a formal demonstration to support it. We only ask you to know how to manipulate a two-dimensional array in PHP and how to perform additions and multiplications on its cells.

Fourthly, when something seems foggy, not very explicit or open to debate in a subject, two options: either you didn't understand, or you must take a position and defend it. The scale will take this into account and keep in mind that a position, even well defended, may be the wrong one.

Lastly, have you ever wondered how **OpenGL** works? Let's find out together today.

Chapter 4

Additional Instructions for today

The following instructions apply to all of today's the exercises. The description in each exercise will serve as a reminder. If you have any doubt, there are no exceptions. These instructions systematically apply whatever way you interpret your reading. Disrespect of one of these instructions leads to a **0** for the exercise and the end of your work's evaluation.

- Only one unique Class per file.
- One file that includes the definition of a class cannot include any other, except for **require** or **require_once** if necessary.
- A file containing a class must **ALWAYS** be named **ClassName.class.php**.
- A class must **ALWAYS** be accompanied by a documentation file which **MUST** be named **ClassName.doc.txt**.
- The documentation of a class must **ALWAYS** be useful and correspond to the implementation. Play by the rules. A class without documentation is useless and flimsy or outdated documentation, does not help either. Be cautious not to explain the internal operation of your classes, the reader of your documentation seeks to know how to use it, and not to understand how you have implemented it. The user can read your code if they want to know this.
- A class must **ALWAYS** have a static Boolean attribute called **verbose** activating the display of useful information for debugging and defence.
- A class must **ALWAYS** have a static method called **doc** that returns the documentation in a string.
- The code that you will write for each exercise, combined with the code from the previous exercises come together toward a building a complete program. To facilitate the development and the defence, the submission of each exercise must also contain a

copy of the files created in previous exercises. However, some exercises will give you the freedom to modify the code of the classes that you have done in previous exercises. In this case, the current exercise will include the modified code instead of the version of the previous exercise, and so on. To conclude, at some point you will also have the freedom to add your own Classes, the corresponding files will of course be include in all submission folder. There is no trick here, don't go nit-picking.

Chapter 5

Exercise 00 : The Color Class

Turn-in directory : ex00/
Files to turn in: Color.class.php, Color.doc.txt
Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library.

Let us start with a simple class: The **Color** Class. This Class will allow us to represent colors and perform a few simple operations on their components.

- The **Color** Class must have three public integer attributes **red**, **green** and **blue** that will be used to represent the components of a color.
- The Class's constructor requires an array. An instance must be able to be built, either by passing a value for the '**rgb**' key which will be split into three red, green and blue components, either by passing a value for the '**red**', '**green**' and '**blue**' keys which will directly represent the three components. Each of the values for the four possible keys will be converted into an integer before use.
- The **Color** Class must have a **__toString** method. See example output for formatting.
- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.
- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.
- The Class must have a static method called **doc** that returns the documentation of the class in a string. (in this specific case the documentation does not have to be long). The content of the documentation must be read from a **Color.doc.txt** file. See example output for formatting the documentation and content of the file.

- The Class must have a method called **add** that allows you to add the the components of the current instance to the components of another instance argument. The resulting color is a new instance.

- The Class must have a method called **sub** that allows you to subtract the components of another instance from the components of the current instance. The resulting color is a new instance.

- The Class must propose a method called **mult** that allows you to multiply the components of the current instance with the components of of another instance argument. The resulting color is a new instance.

In summary, you must write a **Color** Class in a file named **Color.class.php** which allows the **main_00.php** script (attached) to generate the output shown in the **main_00.out** file (also attached). The Class' documentation will be it in a file named **Color.doc.txt**. You will have to submit this file as well.

Chapter 6

Exercise 01 : The Vertex Class

Turn-in directory : ex01/
Files to turn in: Vertex.class.php, Vertex.doc.txt
Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library.

We will now look at the representation of a point in space: the "vertex". We represent a vertex according to five characteristics:

- Its **x** axis coordinate
- Its **y** axis coordinate
- Its **z** depth coordinate
- A new and disturbing coordinate **w**. Search on **Google** "homogeneous coordinates". In practice, this coordinate is often worth **1.0** and will simplify your matrix calculations in the following exercises.
- A color represented by an instance of the **Color** Class from the previous exercise.

The Vertex Class is very simple. It is not yet about understanding the Why and How of a vertex's coordinates. This Class simply offers a coordinate encapsulation and provides reading and writing accessors for the corresponding attributes.

- The **Vertex** Class must possess private attributes to represent the five characteristics (see above). You are reminded that by convention, the identifiers of private attributes begin with the '_' (underscore) character.
- The vertex color will always be an instance of the **Color** Class from the previous exercise.
- The **Vertex** Class must provide reading and writing accessors for its five attributes.

- The Class' constructor is expecting an array. The following keys are required:

  '**x**': x axis coordinate, mandatory.

  '**y**': y axis coordinate, mandatory.

  '**z**': z axis coordinate, mandatory.

  '**w**': optional, by default is worth **1.0**.

  '**color**': optional, by default is worth a new instance of the color white.

- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.

- The **Vertex** Class must have a **__toString** method. See example output for formatting. Please note that the vertex color is added to the string if and only if the static attribute **verbose** is true.

- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.

- The Class must have a static method called **doc** that returns the documentation of the class in a string. (in this specific case the documentation does not have to be long). The content of the documentation must be read from a **Vertex.doc.txt**. From this exercise on, the documentation is left to your discretion. The only requirement is that this documentation is relevant and useful. Play by the rules, imagine that a developer is using your class and they only have your documentation to understand its operation. This point will be evaluated during defence.

In summary, you must write a **Vertex** Class in a file named **Vertex.class.php** which allows the **main_01.php** script (attached) to generate the output shown in the **main_01.out** file (also attached). The Class' documentation will be it in a file named **Vertex.doc.txt**. You will have to submit this file as well.

Chapter 7

Exercise 02: The Vector Class

Turn-in directory : `ex02/`
Files to turn in: `Vector.class.php, Vector.doc.txt`
Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library.

Now that we have completed the vertex, we can place points in space and even give them a simple color. It's cool, but the vectors are practical too, for example, they represent directions or movements.

The **Vector** Class will allow us to introduce a convention. To orient themselves in 3D, one has the choice between a mark called "Left hand" or called "Right hand". Search **Google** for the definition and consider that from now on, we will work in a "Right hand" mark.

If you did your research on homogeneous coordinates when coding the **Vertex** Class, you would have discovered that this representation allow you to drastically simplify some calculations. We will also use a homogeneous system of coordinates for our vectors, but this time, the component **w** will always be worth **0.0** and will considered as an arbitrary vector component in the calculations, like **x**, **y** or **z**.

A vector is represented by the following characteristics:

- Its **x** magnitude
- Its **y** magnitude
- Its **z** magnitude
- The **w** coordinate

The **Vector** Class is barely more complex than the **Vertex** Class. A few methods will ask for very simple calculations that are normally taught in high school. The Internet is full of tutorials on vectors and you only have to adapt them to write this Class.

- The **Vector** Class must possess private attributes to represent the four characteristics (see above). You are reminded that by convention, the identifiers of private attributes begin with the '_' (underscore) character.

- The **Vector** Class must provide read only accessors for its four attributes.

- The Class' constructor is expecting an array. The following keys are required:

  '**dest**': the vector's destination vertex, mandatory.

  '**orig**': the vector's origin vertex, optional, by default is worth a new instance of the **x=0**, **y=0**, **z=0**, **w=1** vertex.

- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.

- The **Vector** Class must have a **__toString** method. See example output for formatting.

- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.

- The Class must have a static method called **doc** that returns the documentation of the class in a string (in this specific case the documentation does not have to be long). The content of the documentation must be read from a **Vector.doc.txt** file and is left to your discretion as stipulated in last previous exercise.

- Methods from the **Vector** Class should never modify the current instance. This behavior is reinforced by the absence of setters. Once a vector is instantiated, its status is final.

- Your **Vector** Class must have at least the following public methods. The existence of private methods is up to you. If some of them seem difficult to code you're probably going too far. Remember, it should only include additions and multiplications.

  float magnitude() : returns the vector's length (or "norm").

Vector normalize[] : returns a normalized version of the vector. If the vector is already normalized, returns a fresh copy of the vector.

Vector add[**Vector $rhs**] : returns the sum vector of both vectors.

Vector sub[**Vector $rhs**] : returns the difference vector of both vectors.

Vector opposite[] : returns the opposite vector.

Vector scalarProduct[**$k**] : returns the multiplication of the vector with a scalar.

float dotProduct[**Vector $rhs**] : returns the scalar multiplication of both vectors.

float cos[**Vector $rhs**] : returns the angle'sAppendix cosine between both vectors.

Vector crossProduct[**Vector $rhs**] : returns the cross multiplication of both vectors (right-hand mark!)

In summary, you must write a **Vector** Class in a file named **Vector.class.php** which allows the **main_02.php** script (attached) to generate the output shown in the **main_02.out** file (also attached). The Class' documentation will be it in a file named **Vector.doc.txt**. You will have to submit this file as well.

Chapter 8

Exercise 03: The Matrix Class

Turn-in directory : `ex03/`
Files to turn in: `Matrix.class.php, Matrix.doc.txt`
Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library.

From this exercise onwards, you will have more freedom when it comes to implementating the requested classes. This means that you can use the attributes and methods which you deem fair, as long as your classes respect the instructions. Be mindful of their visibility. An attribute or a public method with no purpose is an error.

With vertices, one can position points in space. With vectors, it can represent movements in space. With matrices, we will be able to operate transformations, such as apply a scale change, a translation or a rotation to one or several vertices. Often several at once, otherwise game video would be quite boring.

The Internet is overflowing with tutorials on matrices, in particular for 3D. No need to study the entire matrices' mathematical theory, all you need to know here, is: What is a 4x4 matrix? And how to multiply two matrices?

In 3D, a 4x4 matrix can be viewed as the representation of an ortho-standardised mark, namely 3 vectors for the 3 axes and one vertex to the origin of the mark. So we'll represent all our matrices whatever their usefulness in the following manner, namely a basic array:

```
  vtcX vtcY vtcZ vtxO
x    .    .    .    .
y    .    .    .    .
z    .    .    .    .
w    .    .    .    .
```

Under your very eyes, discover below the matrix which represents the mark which each axis vector is the unit vector in its direction and whose origin is the origin vertex:

```
  vtcX vtcY vtcZ vtxO
x  1.0  0.0  0.0  0.0
y  0.0  1.0  0.0  0.0
z  0.0  0.0  1.0  0.0
w  0.0  0.0  0.0  1.0
```

If by chance you have come across a matrix in your life, you should obviously recognized the identity matrix, otherwise say hello.

Take a moment to breathe. If at any time this exercise on matrices seems impossible to understand, it's probably because you focus on the mathematical aspect of the problem instead of the coding. A matrix is an array, nothing more, nothing less. In this exercise, you will need to multiply two matrices and multiply a matrix and a vertex. The algorithms to achieve this are available on the Internet and consist of adding or multiplying two array cells in a certain order. Don't feel obliged to understand why or how these calculations work. Just make sure you apply them.

Our matrices will **ALWAYS** be four rows and four columns big. You can therefore use and abuse this feature in your calculations. To further simplify our problem, our **Matrix** Class will not be used to represent any matrix 4x4. We'll simply settle for the following 4x4 matrices:

- The identity matrix

- The translation matrices ("translate")
- The scale change matrices ("scale")
- The rotation matrices ("rotate")
- the projection matrices ("project")

If you want to learn more about these matrices, **Google** "3D transformation matrices" . If you don't care, simply look at how to build them.

The projection matrix is the most sensitive to calculate. This documentation is perfect to understand it. It gives you the choice to either simply copy the matrix or actually understand what it represents. We will study in more detail its stucture in the next exercise. The other matrices are too simple for us to help you. However, you are encouraged to share your documentation and analysis of the subject.

Let's now define a **Matrix** Class to represent 4x4 matrices. Our matrices will always be of dimension 4x4, no surprises.

- Several behaviors of the **Matrix** Class are to be deducted from the code and of the outputs that follows these explanations. The rest is up to you.
- Your **Matrix** Class must have seven Class constants: **IDENTITY**, **SCALE**, **RX**, **RY**, **RZ**, **TRANSLATION** and **PROJECTION**.
- The Class' constructor is expecting an array. The following keys are required:

  '**preset**': the matrix type to, mandatory. The value must be one of the Class constants previoulsy defined.

  '**scale**': the scale factor, mandatory when '**preset**' is worth **SCALE**.

  '**angle**': the rotation angle in radians, mandatory when '**preset**' is worth **RX**, **RY** or **RZ**.

  '**vtc**': translation vector, mandatory when '**preset**' is worth **TRANSLATION**.

'**fov**': projection field of view in degrees mandatory when '**preset**' is worth **PROJECTION**.

'**ratio**': projected image ratio, mandatory when '**preset**' is worth **PROJECTION**.

'**near**': projection's near clipping plane mandatory when '**preset**' is worth **PROJECTION**.

'**far**': projection's far clipping plane mandatory when '**preset**' is worth **PROJECTION**.

- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.
- The **Matrix** Class must have a **__toString** method. See example output for formatting.
- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.
- The Class must propose a **doc** static method returning a short documentation of the class in a string. The content of the documentation must be read from a **Matrix.doc.txt** file and is left to your discretion as stipulated in previous exercises.
- Methods from the **Matrix** Class should never modify the current instance. Once a matrix is instantiated, its status is final.
- The organization and the code of the **Matrix** Class are up to you. Be smart, efficient and clean. Be careful with the visibility.
- Your **Matrix** Class must have at least the following public methods. If some of them seem too difficult to code you're probably on the wrong track.

Matrix mult[**Matrix $rhs**] : returns the multiplication of both matrices.

Vertex transformVertex[**Vertex $vtx**] : returns a new vertex resulting from the transformation of the vertex by the matrix.

In summary, you must write a **Matrix** Class in a file named **Matrix.class.php** which allows the **main_03.php** script (attached) to generate the output

shown in the **main_03.out** file [also attached]. The Class' documentation will be it in a file named **Matrix.doc.txt**. You will have to submit this file as well.

Chapter 9

Exercise 04: The Camera Class

Turn-in directory : `ex04/`
Files to turn in: `Camera.class.php, Camera.doc.txt, *.class.php, *.doc.txt`
Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library.

From this exercise onward, you are free to modify and enhance the classes from previous exercises if you find it useful. You can also add new classes. However, you must have only one Class per file, respect the file name convention and provide the documentation files that go with your classes. Be careful with the visibility.

Let's review: We're capable of modeling the 3D shapes with the help or our vertices. With our Vectors and our matrices, we can transform these shapes (change their size, move and rotate them). But we are missing something important: the camera to be able to "see" our scenes.

A 3D image is the result of the successive transformation of vertices from a mark to another. We talk about "rendering pipeline" to move from a scene to a displayable image. The OpenGL library pipeline is obviously very complex, for our needs we will use the pipeline below:

```
Local vertex
|
| Model matrix
V
World vertex
|
| View matrix
V
Cam vertex
|
| Projection matrix
V
NDC vertex
|
| Transformation (no matrix involved)
V
Screen vertex (pixel)
```

Usually, we multiply the scale, translation and rotation matrices of a set of vertices to combine these matrises into a resulting matrix called "model matrix". This matrix allows you to transform an object from its local mark toward the "world" mark. This means placing the object wherever we want inthe scene with the desired orientation and size.

To "see" our world, it is necessary to place a camera somewhere in a desired direction. The camera will therefore have coordinates in the "world" mark. To determine the position of the objects in the "camera" mark (what "sees" the camera), it is necessary to compute a matrix which allows you to pass the "world" mark to the the "camera" mark. This matrix is generally called the "view matrix".

How to compute the "view matrix"? This is an excellent question that requires a bit of logic. Let us start by characterizing a camera::

- Its world position is a vertex.
- Its world orientation with a rotation matrix toward "where the camera looks".

The camera's position in the world allows us to compute a translation

matrix. Therefore we have a camera set by a translation matrix and a rotation matrix that allow us to locate the camera in the "world" mark. Now that we have this information, it is possible to compute the "view matrix" by computing the opposite transformation matrix. Take the time to research this concept on the Internet.

- **T** is the translation matrix built from the vector **v**. We compute **oppv v**'s opposite vector and we built an inverse translation matrix **tT**.
- **R** is a rotation matrix. We obtain the matrix **tR** by doing a diagonal symmetry (x become y in the array and vice versa).
- Last step, we multiply tR->mult(tT) and BOOM, we have a "view matrix" for our camera.

At this stage, your camera is able to "see". But the coordinates are always in 3 dimensions. What we would like it is a 2 dimensional image. The scene must therefore be "projected" on a plane. For this, it uses the projection matrix of the previous exercise, defined by the following characteristics:

ratio : The final image ratio, meaning the ratio between the image's width and height. You have heard of 16:9 or 4:3 aspect ration? Well these are image ratios.

fov : The field of view of the projected image in degrees. Look for "3D field of view" on the Internet if you want to learn more. In practice 60 degrees is a correct arbitrary value. It roughly corresponds to the angle between your nose and both edges of the screen of your computer right now. We will be able to modify this value if we want to see a more or less big part of the scene.

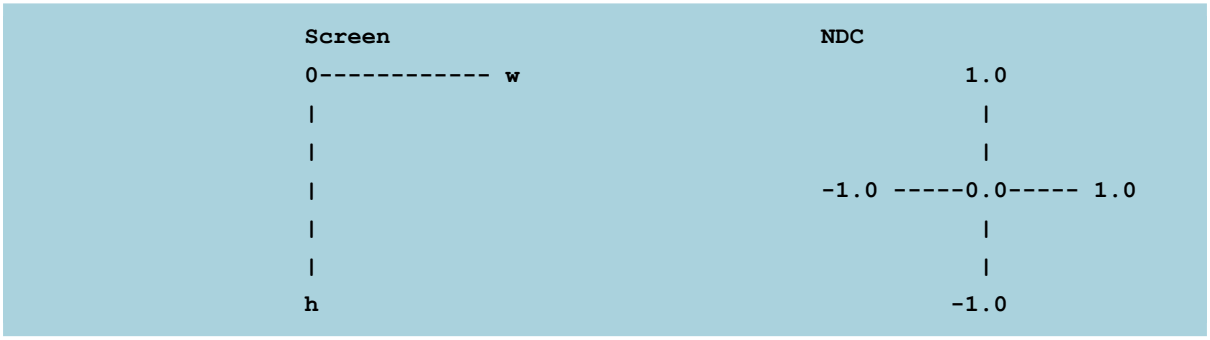
near : The near clipping plane. This concept a little more difficult, to understand. It is the distance of the camera from which an object is seen. **Google** will explain this much better.

far : The far clipping plane. For those interested these two planes enable us to calculate the Z-buffer of a scene, a concept outside the scope of these exercises.

What is certain is that wether you understand how the projection matrix is built or you don't, a camera coordinates' vertex transformed by this matrix will now be in 2D! And 2D implies that we can create an image out of it!

No rush, we're not quite finished yet. The name of the mark in which a vertex is located after being transformed by the projection matrix bears the strange name of "normalized device coordinates", or "NDC" for friends [**Google** it now!]. This mark corresponds to an image in which the Center has for **0.0**, **0.0** coordinates, the lower left corner **-1.0**, **-1.0**, and the upper right corner **1.0**, **1.0**. The **x** and **y** of each vertex are therefore between **-1.0** and **1.0** [the **z** corresponds to the position of the vertex in the Z-buffer [useless for today]]. What is it for? Simply to generate an image of the size you want provided that it respects the ratio! You know the resolution in video games? Well it is how we can change it.

How to move from a NDC vertex to a screen vertex [a pixel]? Think!



Now you'll have to code all of this. Luckily the code is a lot shorter than the text.

- The Class' constructor is expecting an array. The following keys are required:

  '**origin**': The vertex positioning the camera in the world mark. Thanks to this vertex, we can compute a vector and then a translation matrix.

  '**orientation**': Rotation matrix orienting the camera in the world mark.

  '**width**': Width in pixel of the desired image. Is used to compute the ratio. Not compatible with the '**ratio**' key.

  '**height**': Height in pixel of the desired image. Is used to compute the ratio. Not compatible with the '**ratio**' key.

  '**ratio**': Image's ratio. Not compatible with the '**width**' and '**height**' keys.

  '**fov**' : The projected image's field of view in in degree.

  '**near**' : The near clipping plane.

  '**far**' : The far clipping plane.

- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.
- The **Camera** Class must have a **__toString** method. See example output for formatting.
- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.
- The Class must have a static method called **doc** that returns the documentation of the class in a string (in this specific case the documentation doesn not have to be long). The content of the documentation must be read from a **Camera.doc.txt** file and is left to your discretion as stipulated in previous exercises.
- The **Camera** Class's structure and code is up to you.

- Your **Camera** Class must have the following method:

  Vertex watchVertex[**Vertex $worldVertex**] **:** Transforms "world" coordinates vertex into a "screen" coordinates vertex (a pixel basically).

You now have a complete transformation pipeline! Congratulations!

In summary, you must write a **Camera** Class in a file named **Camera.class.php** which allows the **main_04.php** script (attached) to generate the output shown in the **main_04.out** file (also attached). The Class' documentation will be it in a file named **Camera.doc.txt**. You will have to submit this file as well. You're also allowed to modify the previous classes or to add additional ones if you deem it necessary. You must provide a documentation file per additional Class and those Classes must have a **doc** static method like the others.

Chapter 10

Exercise 05: The Triangle and render classes

Turn-in directory : `ex05/`

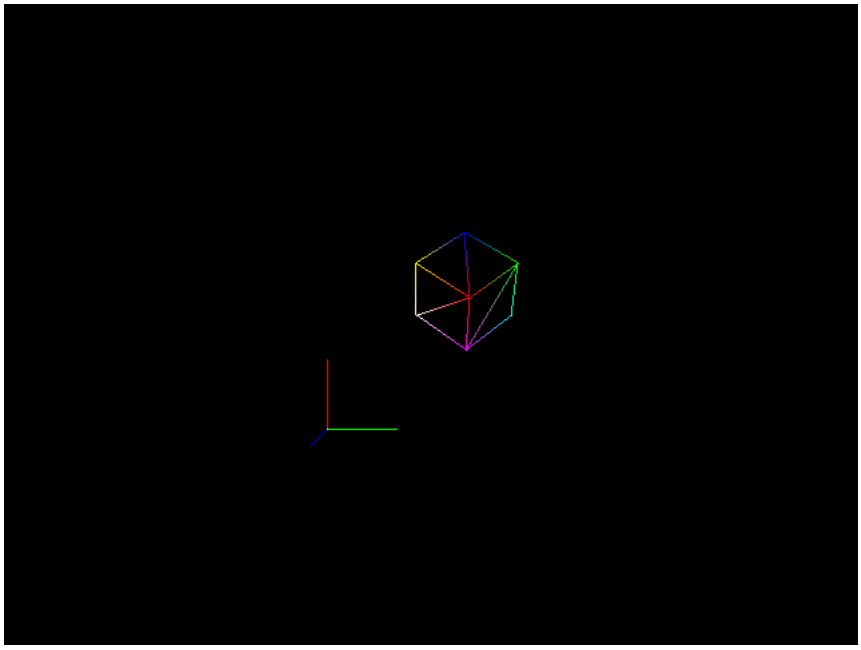
Files to turn in: `Render.class.php, Render.doc.txt, Triangle.class.php, Triangle.doc.txt, *.class.php, *.doc.txt`

Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library and the GD library.

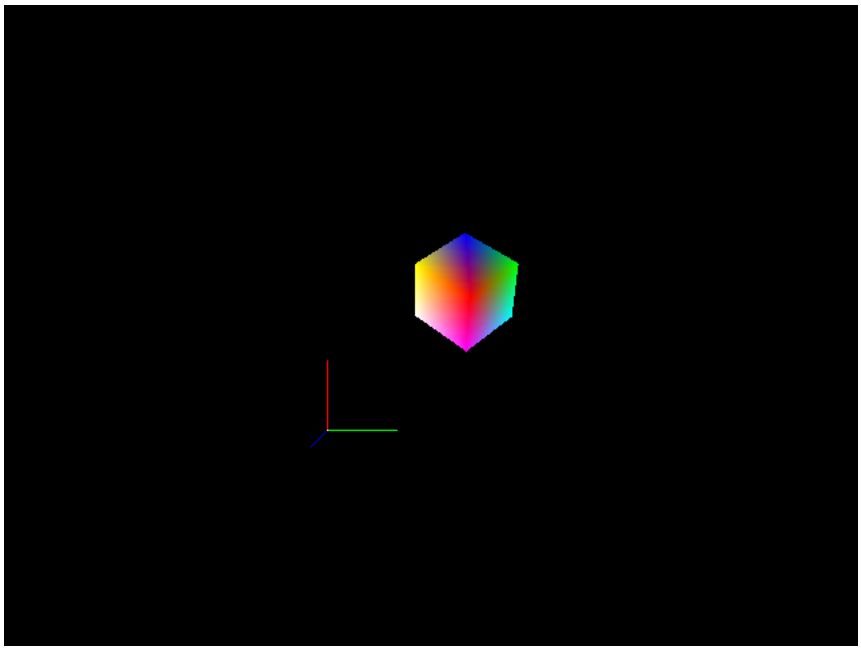
In this exercise, we are finally going to generate an image of our scene. The objective is to be able to render according to 3 modes:

- Vertex :

- Edge :



- Raster :



The generated image must be in **png** format. For this, all the **GD** library of **PHP** is at your disposal.

The implementation of this exercise will be extremely free. You must write a **Triangle** Class as well as **Render** Class. You can add new classes and/or change the classes of the previous exercises. You have to respect the following instructions.

Regarding the **Triangle** Class:

- The **Triangle** Class' constructor is expecting an array. The following keys are required:

  '**A**': Vertex of the first point of the triangle, mandatory.

  '**B**': Vertex of the second point of the triangle, mandatory.

  '**C**': Vertex of the third point of the triangle, mandatory.

- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.
- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.
- The Class must have a static method called **doc** that returns the documentation of the class in a string (in this specific case the documentation doesn not have to be long). The content of the documentation must be read from a **Triangle.doc.txt** file and is left to your discretion as stipulated in previous exercises.
- There are no mandatory methods for your **Triangle** Class. However writing a few of them can be useful. For example I'm thinking about being able to iterate or map the vertices of the triangle, being able to iterate or map the edges of the triangle, be able to sort the vertices or the edges under certain conditions, etc..

Regarding the **Render** Class:

- The **Render** Class' constructor is expecting an array. The following keys are required:

  '**width**': The generated image's width, which is mandatory.

  '**height**': The generated image's height which is mandatory.

  '**filename**': Filename in which the **png** image created will be saved, which is mandatory.

- Your **Render** Class must propose three Class constants: **VERTEX**, **EDGE** and **RASTERIZE** that will be used to select the rendering mode.
- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.
- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.
- The Class must have a static method called **doc** that returns the documentation of the class in a string (in this specific case the documentation doesn not have to be long). The content of the documentation must be read from a **Render.doc.txt** file and is left to your discretion as stipulated in previous exercises.
- Your **Render** Class must have the following methods:

  void renderVertex(**Vertex $screenVertex**) : Displays a "screen" coordinates vertex in the generated image (a pixel basically).

  void renderTriangle(**Triangle $triangle, $mode**) : Displays a "screen" coordinates triangle in the generated image according to the selected mode. The mode must be one of the previoulsy defined Class constants.

  void develop() : Save the **png** generated image on the hard drive using the filename given to the constructor.

To simplify your work, I advise you to add some very simple features to some of the classes of the previous exercises or to add your own classes. This is not mandatory of course. Here are a few ideas without being exhaustive:

- A **toPngColor** method for your **Color** Class for which pseudo code would be the following using **GD** library:

```
color = getColorAlreadyAllocatedInPNGImage( img, r, g, b )
IF color == -1
    IF numberOfColorsInPNGImage( img ) >= 255
        color = getPNGImageClosestColor( img, r, g, b )
```

```
    ELSE
        color = allocateNewColorInPNGImage( img, r, g, b )
RETURN color
```

- Triangle implementation in the **Matrix** and **Camera** Class.
- An additional **bool isVisible(Triangle $tri)** method in the **Camera** Class to determine if a triangle is visible or not to avoid displaying the rear faces of an object. You can use the z-buffer or a BSP if you want but there is a much shorter and simpler algorithm you could implement. Search "backface culling".
- An additional **Mesh** Class to represent a 3D model composed of triangles to allow an easy manipulation of vertices, edges and triangles of the model. Of course, adding the support of this method to the **Matrix** and **Camera** Classes would be extremely useful!
- …

You will find an example of the use of this class in the file **main_05.php** [attached] used to generate the three previous images.

Chapter 11

Bonus exercise 06: The Texture class

Turn-in directory : ex06/

Files to turn in: Texture.class.php, Texture.doc.txt, *.class.php, *.doc.txt, vos fichiers de textures

Allowed functions: Everything learned since the beginning of the Piscine as well as the entire standard PHP library and the GD library.

This exercise if for the strenuous coders. For those of you who wants to score more than 100 for this project. You must implement textures to your rasterizer. You can add and modify everything you want.

Please respect the following instructions:

- The Class must include a Boolean static attribute called **verbose** to control the displays related to the use of the Class. This attribute is initially **False**.
- If and only if the static attribute **verbose** is true, then the Class constructor and destructor will produce an output. See example output for formatting.
- The Class must have a static method called **doc** that returns the documentation of the class in a string (in this specific case the documentation doesn not have to be long). The content of the documentation must be read from a **Texture.doc.txt** file and is left to your discretion as stipulated in previous exercises.

To help you in your research, the useful concepts here are the "**u** and **v** vertex' coordinates" as well as barycentric coordinates of triangle …

Coffee is on me for the first one that succeeds. -- Thor