



# Piscine PHP

Jour 06

Staff 42 [piscine@42.fr](mailto:piscine@42.fr)

*Résumé:*

*Ce document est le sujet du jour 06 de la piscine PHP de 42.*

# Table des matières

<b>I</b>	<b>Consignes</b>	<b>2</b>
<b>II</b>	<b>Préambule</b>	<b>3</b>
<b>III</b>	<b>Introduction</b>	<b>4</b>
<b>IV</b>	<b>Consignes supplémentaires valables toute la journée</b>	<b>6</b>
<b>V</b>	<b>Exercice 00 : La classe Color</b>	<b>8</b>
<b>VI</b>	<b>Exercice 01 : La classe Vertex</b>	<b>10</b>
<b>VII</b>	<b>Exercice 02 : La classe Vector</b>	<b>12</b>
<b>VIII</b>	<b>Exercice 03 : La classe Matrix</b>	<b>15</b>
<b>IX</b>	<b>Exercice 04 : La classe Camera</b>	<b>19</b>
<b>X</b>	<b>Exercice 05 : Les classes Triangle et Render</b>	<b>24</b>
<b>XI</b>	<b>Bonus exercice 06 : La classe Texture</b>	<b>29</b>

# Chapitre I

## Consignes

- Seule cette page servira de référence : ne vous fiez pas aux bruits de couloir.
- Le sujet peut changer jusqu'à une heure avant le rendu.
- Seul le travail rendu sur votre dépôt sera pris en compte par la correction.
- Comme lors de la piscine C, vos exercices seront corrigés par vos camarades de piscine ET par une moulinette.
- La Moulinette est très stricte dans sa notation. Elle est totalement automatisée. Il est impossible de discuter de sa note avec elle. Soyez d'une rigueur irréprochable pour éviter les surprises.
- L'utilisation d'une fonction interdite est un cas de triche. Toute triche est sanctionnée par la note de -42.
- Les exercices sont très précisément ordonnés du plus simple au plus complexe. En aucun cas nous ne porterons attention ni ne prendrons en compte un exercice complexe si un exercice plus simple n'est pas parfaitement réussi.
- Vous ne devez laisser dans votre répertoire aucun autre fichier que ceux explicitement spécifiés par les énoncés des exercices.
- Vous avez une question ? Demandez à votre voisin de droite. Sinon, essayez avec votre voisin de gauche.
- Votre manuel de référence s'appelle **Google**.
- Pensez à discuter sur le forum. La solution à votre problème s'y trouve probablement déjà. Sinon, vous en serez l'instigateur.
- Lisez attentivement les exemples. Ils pourraient bien requérir des choses qui ne sont pas autrement précisées dans le sujet ...
- Réfléchissez. Par pitié, par Thor, par Odin ! Nom d'une pipe !

# Chapitre II

## Préambule

Rien pour le moment.

# Chapitre III

## Introduction

Aujourd'hui débute votre long et passionnant apprentissage de la programmation orientée objet, en particulier avec PHP. Au cours de la semaine passée, vous avez appris les bases syntaxiques et sémantiques de ce langage et nous assumons donc que vous êtes capables d'écrire du code utilisable en PHP.

La couche objet d'un langage a pour principale utilité de faciliter un découpage sémantique de votre code et c'est sur cet aspect en particulier que porteront les exercices d'aujourd'hui. La syntaxe objet de PHP étant très simple, les exercices ne suivront pas un découpage classique d'exercice de piscine, à savoir un exercice == une notion en particulier. Au contraire, les exercices vous guideront à travers la réalisation d'un petit programme amusant, et ce sera à vous de déduire des informations qui vous sont données quelles notions sont utiles à la réalisation de l'exercice. Oui, il va falloir réfléchir et confronter vos idées.

Pour ceux d'entre-vous ayant de l'expérience en POO, vous n'aurez pas manqué de remarquer que les vidéos du jour ne traitent qu'une partie de la POO en PHP, la partie programmation modulaire pour être précis. Bien entendu, le reste des notions seront abordées demain et après demain. En ce qui concerne les exercices d'aujourd'hui, vous ne DEVEZ utiliser que ce qui a été vu jusqu'à ce jour. Pas d'exceptions, d'héritages, traits et autres interfaces. Ca sera pour demain et après demain.

Au cours de cette piscine, et dans les projets à venir, vous aurez votre lot de sites et autres applications web à réaliser. Alors pourquoi ne pas profiter de cette journée pour faire autre chose, comme de la 3D par exemple ? Vous avez fait un Raytracer il n'y a pas si longtemps, maintenant on va faire un Rasterizer.

Plusieurs choses à dire avant de commencer. Premièrement, cette journée de piscine sera uniquement évaluée par peer-correction. Donc pas d'angoisse de moulinette à avoir. Toutes les sorties d'exemple ne sont pas rigoureusement à respecter à l'espace ou au pixel près, même si les énoncés encouragent un certain formatage que je vous invite à respecter. Ce qui nous intéresse c'est le résultat et l'organisation de votre code.

Deuxièmement un certain nombre de recherches de votre part sont nécessaires pour aborder les exercices du jour. Autant les notions techniques liées à PHP sont décrites dans les vidéos accompagnant ce sujet, autant le vocabulaire et les notions liées à la 3D

sont à découvrir par vous même.

Troisièmement, contrairement à ce que certains énoncés peuvent laisser supposer, les notions mathématiques à connaître pour réaliser ce jour de piscine sont infimes. Si vous vous perdez dans les mathématiques c'est que vous vous y prenez mal. Je ne vous demande pas de savoir contruire une matrice de projection devant un amphi avec explication de tous les details de transformation, démonstration formelle à l'appui. Je vous demande de savoir manipuler un tableau à deux dimensions en PHP et de faire des additions et des multiplications sur ses cases.

Quatrièmement, lorsque quelque chose vous parait flou, pas très explicite ou sujet à débat dans un énoncé, deux solutions : soit vous n'avez pas compris, soit vous devez prendre une décision et la défendre. Le barème en tiendra compte et gardez à l'esprit qu'une décision, même bien défendue, peut être mauvaise.

Dernièrement, vous-êtes vous déjà demandé comment fonctionne `OpenGL` ? Eh bien découvrons-le ensemble aujourd'hui.

# Chapitre IV

## Consignes supplémentaires valables toute la journée

Les consignes suivantes s'appliquent à tous les exercices de la journée. La description de chaque exercice les rappellera. Si vous deviez avoir un doute, il n'y a jamais d'exception. Ces consignes s'appliquent systématiquement quoi que vous puissiez interpréter de votre lecture. Le non respect d'une de ces consignes entraîne 0 à l'exercice et la fin de l'évaluation de votre travail sur le barème.


- Une seule et unique classe par fichier.
- Un fichier qui contient la définition d'une classe ne doit rien contenir d'autre, à part des `require` ou `require_once` quand c'est nécessaire.
- Un fichier contenant une classe doit **TOUJOURS** être nommé sous la forme `ClassName.class.php`.
- Une classe doit **TOUJOURS** être accompagnée d'un fichier de documentation dont le nom doit **TOUJOURS** être de la forme `ClassName.doc.txt`.
- La documentation d'une classe doit **TOUJOURS** être utile et correspondre à l'implémentation. Jouez le jeu. Une classe sans documentation ne sert à rien et une documentation bancale, obsolète ou incomplète n'aide pas plus. Faites également attention à ne pas expliquer le fonctionnement interne de vos classes. Le lecteur de votre documentation cherche à savoir l'utiliser pas à comprendre comment vous l'avez implémentée. Pour cela il peut lire votre code.
- Une classe doit **TOUJOURS** proposer un attribut statique boolean `verbose` permettant d'afficher des informations pratiques pour le debug et la soutenance.
- Une classe doit **TOUJOURS** proposer une méthode statique `doc` renvoyant le contenu du fichier de documentation associé à une classe sous forme d'une chaîne de caractères.
- Le code que vous allez écrire pour chaque exercice s'ajoute à celui des exercices précédents pour converger vers un programme complet. Pour faciliter le développement et la soutenance, le rendu de chaque exercice doit également contenir une copie des fichiers rendus dans les exercices précédents. Cependant, certains exercices vous laissent la liberté de modifier le code des classes que vous avez réalisées dans les exercices précédents. Dans ce cas, l'exercice courant inclura bien évidemment

le code modifié à la place de la version de l'exercice précédent, et ainsi de suite. Pour terminer, vous aurez également la liberté d'ajouter vos propres classes à partir d'un moment, les fichiers correspondant devront bien évidemment être présents d'un dossier de rendu à l'autre. Il n'y a pas de piège, ne cherchez pas la petite bête.



# Chapitre V

## Exercice 00 : La classe Color

	Exercice : 00
La class Color	
Dossier de rendu : <i>ex00/</i>	
Fichiers à rendre : <code>Color.class.php</code> , <code>Color.doc.txt</code>	
Fonctions Autorisées : Tout ce que qui a été vu depuis le début de la piscine et toute la bibliothèque standard de PHP.	
Remarques : n/a	

Commençons par une petite classe toute simple : la classe `Color`. Cette classe va nous permettre de représenter des couleurs et de faire quelques opérations simples sur leurs composantes.

- La classe `Color` doit posséder trois attributs publics entiers `red`, `green` et `blue` qui vous serviront à représenter les composantes d'une couleur.
- Le constructeur de la classe attend un tableau. Une instance doit pouvoir être construite, soit en passant une valeur pour la clef `'rgb'` qui sera décomposée en trois composantes rouge, verte et bleue, soit en passant une valeur pour les clefs `'red'`, `'green'` et `'blue'` qui représenteront directement les trois composantes. Chacune des valeurs pour les quatre clefs possibles sera convertie en entier avant utilisation.
- La classe `Color` doit proposer une méthode `__toString`. Voir la sortie d'exemple pour déduire le formatage.
- La classe doit contenir un attribut **verbose statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur `False`.
- Si et seulement si l'attribut statique `verbose` est vrai, alors le constructeur et le destructeur de la classe produiront une sortie. Voir la sortie d'exemple pour déduire le formatage.
- La classe doit proposer une méthode statique `doc` retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documen-


tation devra être lu depuis un fichier `Color.doc.txt`. Voir la sortie d'exemple pour déduire le formatage de la documentation et le contenu du fichier.

- La classe doit proposer une méthode `add` permettant d'ajouter les composantes de l'instance courante aux composantes d'une instance passée en paramètre. La couleur résultat est une nouvelle instance.
- La classe doit proposer une méthode `sub` permettant de soustraire les composantes d'une instance passée en paramètre aux composantes de l'instance courante. La couleur résultat est une nouvelle instance.
- La classe doit proposer une méthode `mult` permettant de multiplier les composantes de l'instance courante par un facteur passé en paramètre. La couleur résultat est une nouvelle instance.

Pour résumer formellement, vous devez écrire une classe `Color` dans un fichier à rendre nommé `Color.class.php` qui permet au script `main_00.php` donné en annexe de ce sujet de générer la sortie présente dans le fichier `main_00.out` également donné en annexes de ce sujet. La documentation de la classe sera elle dans un fichier nommé `Color.doc.txt`, fichier à rendre lui aussi.

# Chapitre VI

## Exercice 01 : La classe Vertex

	Exercice : 01
La classe Vertex	
Dossier de rendu : <i>ex01/</i>	
Fichiers à rendre : <code>Vertex.class.php</code> , <code>Vertex.doc.txt</code>	
Fonctions Autorisées : Tout ce que qui a été vu depuis le début de la piscine et toute la bibliothèque standard de PHP.	
Remarques : n/a	

Nous allons maintenant nous intéresser à la représentation d'un point dans l'espace : le "vertex". Nous représenterons un vertex selon cinq caractéristiques :

- Son abscisse **x**
- Son ordonnée **y**
- Sa profondeur **z**
- Une coordonnée nouvelle et inquiétante **w**. Recherchez sur **Google** "coordonnées homogènes". En pratique, cette coordonnée vaudra souvent 1.0 et simplifiera vos calculs matriciels dans les exercices suivants.
- Une couleur représentée sous la forme d'une instance de la classe **Color** de l'exercice précédent.

La classe **Vertex** est très simple. Il n'est pas encore question de comprendre le pourquoi du comment des coordonnées d'un vertex. Cette classe propose simplement une encapsulation aux coordonnées et fournit les accesseurs en lecture et en écriture pour les attributs correspondant.

- La classe **Vertex** doit posséder des attributs privés pour représenter les cinq caractéristiques précédentes. Je vous rappelle que par convention, les identifiants des attributs privés commencent par le caractère `'_'` (underscore).
- La couleur du vertex sera toujours une instance de la classe **Color** de l'exercice


précédent.

- La classe **Vertex** doit fournir des accesseurs en lecture et en écriture pour ses cinq attributs.
- Le constructeur de la classe attend un tableau. Les clefs attendues sont les suivantes :
  - 'x' : l'abscisse du vertex, obligatoire.
  - 'y' : l'abscisse du vertex, obligatoire.
  - 'z' : l'abscisse du vertex, obligatoire.
  - 'w' : optionnel, vaut 1.0 par défaut.
  - 'color' : optionnel, vaut une instance fraîche de la couleur blanche par défaut.
- La classe doit contenir un attribut **verbose statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur **False**.
- La classe **Vertex** doit proposer une méthode **\_\_toString**. Voir la sortie d'exemple pour déduire le formatage. Notez que la couleur du vertex n'est ajoutée à la string que si et seulement si l'attribut statique **verbose** est vrai.
- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie. Voir la sortie d'exemple pour déduire le formatage.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier **Vertex.doc.txt**. A partir de cet exercice, la documentation est laissée à votre discrétion. La seule obligation est que cette documentation soit pertinente et utile. Jouez donc le jeu, imaginez un développeur devant utiliser votre classe et n'ayant que cette documentation pour comprendre son fonctionnement. Ce point sera bien évidemment vérifié en soutenance.

Pour résumer formellement, vous devez écrire une classe **Vertex** dans un fichier à rendre nommé **Vertex.class.php** qui permet au script **main\_01.php** donné en annexes de ce sujet de générer la sortie présente dans le fichier **main\_01.out** également donné en annexes de ce sujet. La documentation de la classe sera elle dans un fichier nommé **Vertex.doc.txt**, fichier à rendre lui aussi.

# Chapitre VII

## Exercice 02 : La classe Vector

	Exercice : 02
La classe Vector	
Dossier de rendu : <i>ex02/</i>	
Fichiers à rendre : <code>Vector.class.php</code> , <code>Vector.doc.txt</code>	
Fonctions Autorisées : Tout ce que qui a été vu depuis le début de la piscine et toute la bibliothèque standard de PHP.	
Remarques : n/a	

Maintenant qu'on a des vertices, on peut placer des points dans l'espace et même leur donner une couleur simple. C'est chouette, mais les vecteurs c'est pratique aussi, pour représenter des directions ou des déplacements par exemple.

La classe **Vector** va nous permettre d'introduire une convention. Pour s'orienter en 3D, on a le choix entre un repère dit "main gauche" ou dit "main droite". Recherchez sur **Google** ce que cela signifie et considérez qu'à partir de maintenant, nous travaillerons dans un repère "main droite".

Si vous vous êtes renseignés sur les coordonnées homogènes en réalisant la classe **Vertex**, vous avez découvert que cette représentation permet de simplifier énormément certains calculs. Nous allons également utiliser un système de coordonnées homogènes pour nos vecteurs, mais cette fois, la composante **w** vaudra toujours 0.0 et sera considérée comme une composante quelconque du vecteur dans les calculs, au même titre que **x**, **y** ou **z**.

Un vecteur est représenté par les caractéristiques suivantes :

- Sa magnitude en **x**
- Sa magnitude en **y**
- Sa magnitude en **z**
- La coordonnée **w**

La classe **Vector** est à peine plus complexe que la classe **Vertex**. Quelques méthodes demanderont des calculs très simples qui sont normalement abordés en seconde au lycée. Le net regorge bien évidemment de tutoriels sur les vecteurs et vous n'aurez qu'à les adapter pour écrire cette classe.

- La classe **Vector** doit posséder des attributs privés pour représenter les quatre caractéristiques précédentes. Je vous rappelle que par convention, les identifiants des attributs privés commencent par le caractère '\_' (underscore).
- La classe **Vector** doit fournir des accesseurs en lecture seule pour ses quatre attributs.
- Le constructeur de la classe attend un tableau. Les clefs attendues sont les suivantes :  
**'dest'** : vertex de destination du vecteur, obligatoire.  
**'orig'** : vertex d'origine du vecteur, optionnel, vaut une instance fraîche du vertex `x=0, y=0, z=0, w=1` par défaut.
- La classe doit contenir un attribut **verbose statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur **False**.
- La classe **Vector** doit proposer une méthode `__toString`. Voir la sortie d'exemple pour déduire le formatage.
- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie. Voir la sortie d'exemple pour déduire le formatage.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier `Vector.doc.txt` et est laissée à votre discrétion, comme dans l'exercice précédent.
- Une méthode de la classe **Vector** ne doit jamais modifier l'instance courante. Ce comportement est renforcé par l'absence de setters. Une fois un vecteur instancié, son état est définitif.
- votre classe **Vector** doit proposer au moins les méthodes publiques suivantes. L'existence de méthodes privées ne regarde que vous. Si certaines vous paraissent difficiles à coder, c'est que vous vous prenez la tête. Il ne s'agit que d'additions et de multiplications.

**float magnitude()** : retourne la longueur (ou "norme") du vecteur.

**Vector normalize()** : retourne le vecteur normalisé. Si le vecteur est déjà normalisé, retourne une copie fraîche du vecteur.

**Vector add( Vector \$rhs )** : retourne le vecteur somme des deux vecteurs.

**Vector sub( Vector \$rhs )** : retourne le vecteur difference des deux vecteurs.

**Vector opposite()** : retourne le vecteur opposé.

**Vector scalarProduct( \$k )** : retourne le produit du vecteur avec un scalaire.

**float dotProduct( Vector \$rhs )** : retourne le produit scalaire de deux vecteurs.


**float cos( Vector \$rhs )** : retourne le cosinus de l'angle entre deux vecteurs.

**Vector crossProduct( Vector \$rhs )** : retourne le produit en croix de deux vecteurs (repère main droite!)

Pour résumer formellement, vous devez écrire une classe **Vector** dans un fichier à rendre nommé **Vector.class.php** qui permet au script **main\_02.php** donné en annexe de ce sujet de générer la sortie présente dans le fichier **main\_02.out** également donné en annexes de ce sujet. La documentation de la classe sera elle dans un fichier nommé **Vector.doc.txt**, fichier à rendre lui aussi.

# Chapitre VIII

## Exercice 03 : La classe Matrix

	Exercice : 03
La classe Matrix	
Dossier de rendu : <i>ex03/</i>	
Fichiers à rendre : <code>Matrix.class.php</code> , <code>Matrix.doc.txt</code>	
Fonctions Autorisées : Tout ce que qui a été vu depuis le début de la piscine et toute la bibliothèque standard de PHP.	
Remarques : n/a	

A partir de cet exercice, l'implémentation des classes demandées va devenir de plus en plus libre. Cela signifie que vous pouvez utiliser les attributs et les méthodes qui vous semblent justes tant que vos classes répondent aux consignes. Soyez soucieux de leur visibilité. Un attribut ou une méthode publique qui n'a pas lieu d'être est une erreur.

Avec des vertices, on peut positionner des points dans l'espace. Avec des vecteurs, on peut représenter des déplacements dans l'espace. Avec des matrices, on va pouvoir commencer à faire des transformations, comme appliquer un changement d'échelle, une translation ou une rotation à un ou plusieurs vertices. Souvent plusieurs, sinon le jeu video ça serait chiant.

Internet est plein à craquer de tutoriels sur les matrices, en particulier pour le domaine de la 3D. Inutile d'apprendre toute la théorie mathématique des matrices, ce dont on a besoin de savoir ici, c'est : qu'est ce qu'une matrice 4x4 ? Et comment on multiplie deux matrices ?



En 3D, une matrice 4x4 peut être vue comme la représentation d'un repère orthonormé, à savoir 3 vecteurs pour les 3 axes et un vertex pour l'origine du repère. Nous représenterons donc toutes nos matrices quelque soit leur utilité de la manière suivante, à savoir un bête tableau :

```

vtxX vtxY vtxZ vtx0
x   .   .   .   .
y   .   .   .   .
z   .   .   .   .
w   .   .   .   .

```

Sous vos yeux ébahis, découvrez ci-dessous la matrice qui représente le repère dont chaque vecteur d'axe est le vecteur unité dans sa direction et dont l'origine est le vertex origine :

```

vtxX vtxY vtxZ vtx0
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

```

Si par hasard vous avez déjà croisé une matrice dans votre vie, vous venez bien entendu de reconnaître la matrice identité, sinon, eh bien dites bonjour.

Prenez un instant pour respirer. Si à un moment cet exercice sur les matrices vous semble impossible à comprendre, c'est que vous vous concentrez sur l'aspect mathématique du problème au lieu de vous concentrer sur l'aspect informatique. Une matrice, c'est un tableau, rien de plus, rien de moins. Dans cet exercice, vous allez devoir multiplier deux matrices ensemble et multiplier une matrice et un vertex. Les algos pour y parvenir sont disponibles en très grand nombre sur internet et consistent juste à additionner ou multiplier deux cases d'un tableau dans un certain ordre. Ne vous sentez pas obligés de comprendre pourquoi ou comment ces calculs fonctionnent. Appliquez les.

Nos matrices feront **TOUJOURS** quatre lignes et quatre colonnes. Vous pouvez donc user et abuser de cette caractéristique dans vos calculs. Pour simplifier encore notre problème, notre classe **Matrix** ne servira pas à représenter n'importe quelle matrice 4x4. Nous nous contenterons des matrices 4x4 suivantes :

- La matrice identité
- Les matrices de translation ("translate")
- Les matrices de changement d'échelle ("scale")
- Les matrices de rotation ("rotate")
- Les matrices de projection ("project")

Si vous voulez en savoir plus sur ces matrices, recherchez les "matrices de transformation 3D" sur [Google](#). Si vous vous en fichez, contentez vous de chercher comment les construire.

La matrice de projection est la plus délicate à calculer. Cette [documentation](#) est parfaite pour vous y retrouver. Elle vous laisse le choix de recopier la matrice bêtement ou de comprendre ce qu'elle représente. Nous étudierons sa composition plus en détails dans l'exercice suivant. Les autres matrices sont trop simples pour que je vous aide. Toutefois je vous encourage à partager vos documentations et analyses de la question.

Définissons maintenant une classe **Matrix** pour représenter des matrices 4x4. Nos matrices seront toujours de dimension 4x4, pas de surprises.

- Plusieurs comportements de la classe **Matrix** sont à déduire du code et de la sortie qui suivent ces explications. Le reste est à votre discrétion.
- Votre classe **Matrix** doit proposer sept constantes de classe : **IDENTITY**, **SCALE**, **RX**, **RY**, **RZ**, **TRANSLATION** et **PROJECTION**.
- Le constructeur de la classe attend un tableau. Les clefs attendues sont les suivantes :
  - 'preset' : type de matrice à construire, obligatoire. La valeur doit être l'une des constantes de classe précédentes.
  - 'scale' : le facteur d'échelle, obligatoire quand 'preset' vaut **SCALE**.
  - 'angle' : angle de rotation en radians, obligatoire quand 'preset' vaut **RX**, **RY** ou **RZ**.
  - 'vtc' : vecteur de translation, obligatoire quand 'preset' vaut **TRANSLATION**.
  - 'fov' : champs de vision de la projection en degrés, obligatoire quand 'preset' vaut **PROJECTION**.
  - 'ratio' : ratio de l'image projetée, obligatoire quand 'preset' vaut **PROJECTION**.
  - 'near' : plan de clipping proche de la projection, obligatoire quand 'preset' vaut **PROJECTION**.
  - 'far' : plan de clipping éloigné de la projection, obligatoire quand 'preset' vaut **PROJECTION**.
- La classe doit contenir un attribut **verbose** **statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur **False**.
- La classe **Matrix** doit proposer une méthode **\_\_toString**. Voir la sortie d'exemple pour déduire le formatage.
- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie. Voir la sortie d'exemple pour déduire le formatage.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier **Matrix.doc.txt** et est laissée à votre discrétion, comme dans l'exercice précédent.

- Une méthode de la classe **Matrix** ne doit jamais modifier l'instance courante. Une fois une matrice instanciée, son état est définitif.
- l'organisation et le code de la classe **Matrix** sont à votre discrétion. Soyez malins, efficaces et propres. Soyez soigneux avec la visibilité.
- votre classe **Matrix** doit proposer les méthodes suivantes. Si certaines vous paraissent difficiles à coder, c'est que vous vous prenez la tête ou bien que vous vous y prenez mal.


**Matrix mult( Matrix \$rhs )** : retourne une nouvelle matrice résultat de la multiplication des deux matrices.

**Vertex transformVertex( Vertex \$vtx )** : retourne un nouveau vertex résultat de la transformation du vertex par la matrice.

Pour résumer formellement, vous devez écrire une classe **Matrix** dans un fichier à rendre nommé **Matrix.class.php** qui permet au script **main\_03.php** donné en annexe de ce sujet de générer la sortie présente dans le fichier **main\_03.out** également donné en annexes de ce sujet. La documentation de la classe sera elle dans un fichier nommé **Matrix.doc.txt**, fichier à rendre lui aussi.

# Chapitre IX

## Exercice 04 : La classe Camera

	Exercice : 04
La classe Camera	
Dossier de rendu : <i>ex04/</i>	
Fichiers à rendre : <code>Camera.class.php</code> , <code>Camera.doc.txt</code> , <code>*.class.php</code> , <code>*.doc.txt</code>	
Fonctions Autorisées : Tout ce que qui a été vu depuis le début de la piscine et toute la bibliothèque standard de PHP.	
Remarques : n/a	

A partir de cet exercice, vous êtes libres de modifier et de compléter les classes des exercices précédents si vous en voyez l'utilité. Vous pouvez également ajouter de nouvelles classes. Vous devez toutefois n'avoir qu'une seule classe par fichier, respecter les conventions de noms de fichiers et fournir le fichier de documentation qui va avec vos classes. Soyez soigneux avec la visibilité.

Faisons le point : On est capable de modéliser des formes en 3D à l'aide de nos vectices. Avec nos vecteurs et nos matrices, on peut transformer ces formes (changer leur taille, les déplacer et les tourner). Il nous manque quelque chose d'important : la caméra pour pouvoir "voir" nos scènes.

Une image 3D est le résultat de la transformation successive de vertices d'un repère à l'autre. On parle de "pipeline de transformations" pour passer d'une scène à une image affichable. Le pipeline d'une bibliothèque comme `OpenGL` étant bien entendu très complexe, nous utiliserons ce pipeline largement suffisant pour nos besoins :

```
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)
```

Généralement, on combine par multiplication les matrices d'échelle, de translation et de rotation d'un ensemble de vertices en une matrice résultat appelée "model matrix". Cette matrice permet de transformer un objet depuis son repère local vers le repère du "monde". Cela signifie en gros placer l'objet où on veut dans la scène avec l'orientation et la taille voulue.

Pour "voir" notre monde, il est nécessaire de placer une caméra quelque part selon une orientation voulue. La caméra aura donc des coordonnées dans le repère "monde". Pour déterminer la position des objets dans le repère "caméra" (ce que "voit" la caméra), il faut calculer une matrice qui permet de passer du repère "monde" vers le repère "caméra". Cette matrice s'appelle généralement la "view matrix".

Comment calculer la "view matrix"? Excellente question qui nécessite un peu de logique. Commençons déjà par caractériser une caméra :

- Sa position dans le monde avec un vertex.
- Son orientation dans le monde avec une matrice de rotation, c'est à dire vers "où la caméra regarde".

La position de la caméra dans le monde nous permet de calculer une matrice de translation. Nous avons donc une caméra définie par une matrice de translation et une matrice de rotation qui permettent de situer la caméra dans le repère "monde". Maintenant que nous avons ces informations, calculer la "view matrix" devient possible en calculant la matrice de transformation inverse. Prenez le temps de chercher sur le net.

- Soit  $T$  la matrice de translation construite à partir du vecteur  $v$ . On calcule  $oppv$  le vecteur opposé de  $v$  et on construit une matrice de translation inverse  $tT$ .
- Soit la matrice de rotation  $R$ . On fait une symétrie diagonale (les  $x$  deviennent les  $y$  dans le tableau et vice versa) et on obtient une matrice  $tR$ .
- Dernière étape, on multiplie  $tR \rightarrow mult(tT)$  et paf, ça fait une "view matrix" pour votre caméra.

A ce stade, votre caméra est capable de "voir". Mais les coordonnées sont toujours en 3 dimensions, alors que ce qu'on voudrait c'est une image en 2 dimensions. La scène regardée doit donc être "projetée" sur un plan. Pour cela, on utilise la matrice de projection de l'exercice précédent définie par les caractéristiques suivantes :

**ratio** : Le ratio de l'image finale, c'est à dire le rapport entre la largeur et la hauteur de l'image. Vous avez entendu parler de format quatre tiers ou seize neuvièmes ? Eh bien ce sont des ratios d'images.

**fov** : Le champs de vision de l'image projetée en degrés. Cherchez "field of view 3D" sur [Google](#) si vous voulez en savoir plus. En pratique 60 degrés est une valeur arbitraire correcte. Elle correspond à peu près à l'angle entre votre nez et les deux bords de l'écran de votre ordinateur en ce moment même. On pourra modifier cette valeur pour voir une portion plus ou moins grande de la scène.

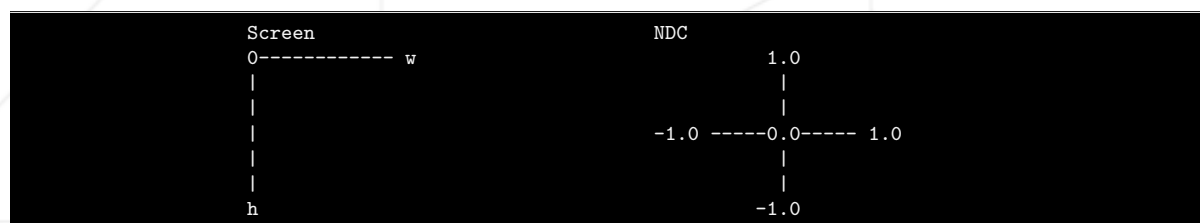
**near** : Le plan de clipping proche. Notion un peu complexe, c'est la distance de la caméra à partir de laquelle un objet est vu. [Google](#) vous expliquera.

**far** : Le plan de clipping éloigné. Pour ceux que ça intéresse, ces deux plans permettent de calculer le z-buffer d'une scène, notion hors du scope de ces exercices.

Ce qui est certain, c'est que vous compreniez comment est construite la matrice de projection ou non, un vertex en coordonnées caméra transformé par cette matrice sera maintenant en 2D ! Et qui dit 2D dit qu'on peut en faire une image !

Pas de précipitations, on a pas tout à fait fini. Le nom du repère dans lequel se trouve un vertex après transformation par la matrice de projection porte l'étrange nom de "normalized device coordinates", ou "NDC" pour les intimes ([Google](#) it now !). Ce repère correspond à une image dont le centre a pour coordonnées 0.0, 0.0, le coin inférieur gauche -1.0, -1.0 et le coin supérieur droit 1.0, 1.0. Les x et les y de chaque vertex sont donc compris entre -1.0 et 1.0 (le z correspond à la position du vertex dans le z-buffer (inutile pour aujourd'hui). A quoi ça sert ? Et bien tout simplement à générer une image de la taille que vous voulez pourvu qu'elle respecte le ratio ! Vous savez la résolution dans les jeux vidéo ? Et bien c'est comme ça qu'on fait pour la changer.

Comment passer d'un vertex NDC à un vertex écran (un pixel) ? Réfléchissez !



Il va falloir coder tout ça maintenant. Heureusement, le code est bien plus court que ce texte.

- Le constructeur de la classe attend un tableau. Les clefs attendues sont les suivantes :
  - '**origin**' : Vertex positionnant la caméra dans le repère monde. Grâce à ce vertex, on peut calculer un vecteur puis une matrice de translation.
  - '**orientation**' : Matrice de rotation orientant la caméra dans le repère monde.
  - '**width**' : Largeur en pixels de l'image voulue. Sert à calculer le ratio. Incompatible avec la clef '**ratio**'.
  - '**height**' : Hauteur en pixels de l'image voulue. Sert à calculer le ratio. Incompatible avec la clef '**ratio**'.
  - '**ratio**' : Ratio de l'image. Incompatible avec les clefs '**width**' et '**height**'.
  - '**fov**' : Le champs de vision de l'image projetée en degrés.
  - '**near**' : Le plan de clipping proche.
  - '**far**' : Le plan de clipping éloigné.
- La classe doit contenir un attribut **verbose** **statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur **False**.
- La classe **Camera** doit proposer une méthode **\_\_toString**. Voir la sortie d'exemple pour déduire le formatage.
- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie. Voir la sortie d'exemple pour déduire le formatage.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier **Camera.doc.txt** et est laissée à votre discrétion, comme dans les exercices précédents.
- l'organisation et le code de la classe **Camera** sont à votre discrétion.

- votre classe **Camera** doit proposer la méthode suivante :

**Vertex watchVertex( Vertex \$worldVertex )** : Transforme un vertex en coordonnées "monde" en un vertex en coordonnées "écran" (en pixel quoi).


Vous avez à présent un pipeline de transformation complet ! Félicitations !

Pour résumer formellement, vous devez écrire une classe **Camera** dans un fichier à rendre nommé **Camera.class.php** qui permet au script **main\_04.php** donné en annexes de ce sujet de générer la sortie présente dans le fichier **main\_04.out** également donné en annexes de ce sujet. La documentation de la classe sera elle dans un fichier nommé **Camera.doc.txt**, fichier à rendre lui aussi. Vous avez également le droit de modifier les classes précédentes ou d'en ajouter si vous le jugez nécessaire. Vous devez fournir un fichier de doc par classe supplémentaire et ces classes doivent avoir la méthode statique **doc** comme les autres.



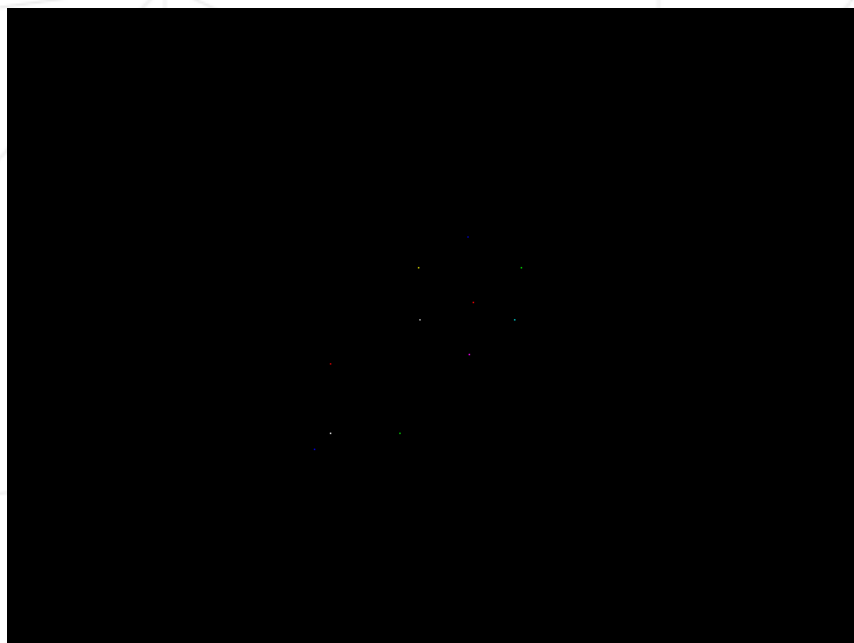
# Chapitre X

## Exercice 05 : Les classes Triangle et Render

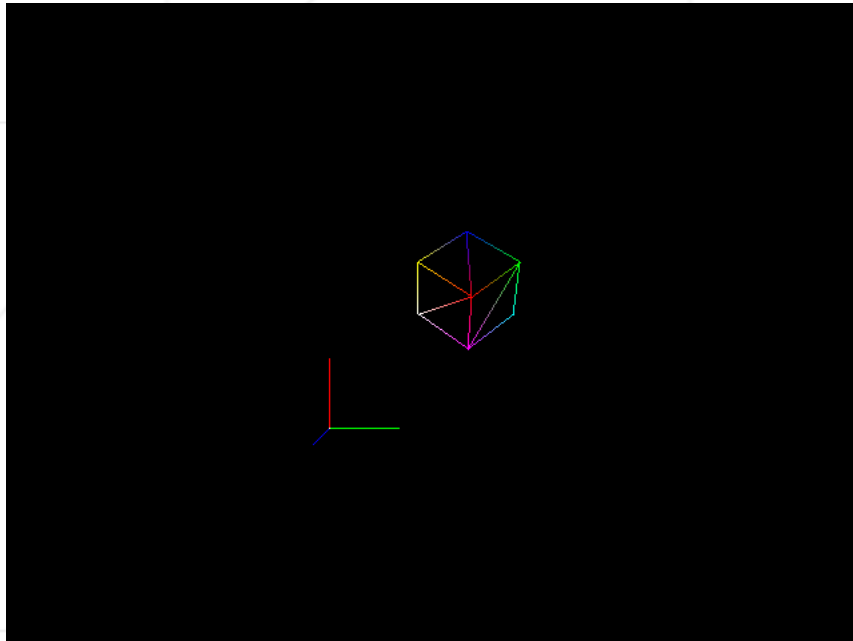
	Exercice : 05
Les classes Triangle et Render	
Dossier de rendu : <i>ex05/</i>	
Fichiers à rendre : <code>Render.class.php</code> , <code>Render.doc.txt</code> , <code>Triangle.class.php</code> , <code>Triangle.doc.txt</code> , <code>*.class.php</code> , <code>*.doc.txt</code>	
Fonctions Autorisées : Tout ce qui a été vu depuis le début de la piscine, toute la bibliothèque standard de PHP et la bibliothèque GD.	
Remarques : n/a	

Dans cet exercice, nous allons enfin générer une image de notre scène. L'objectif est de pouvoir rendre selon 3 modes :

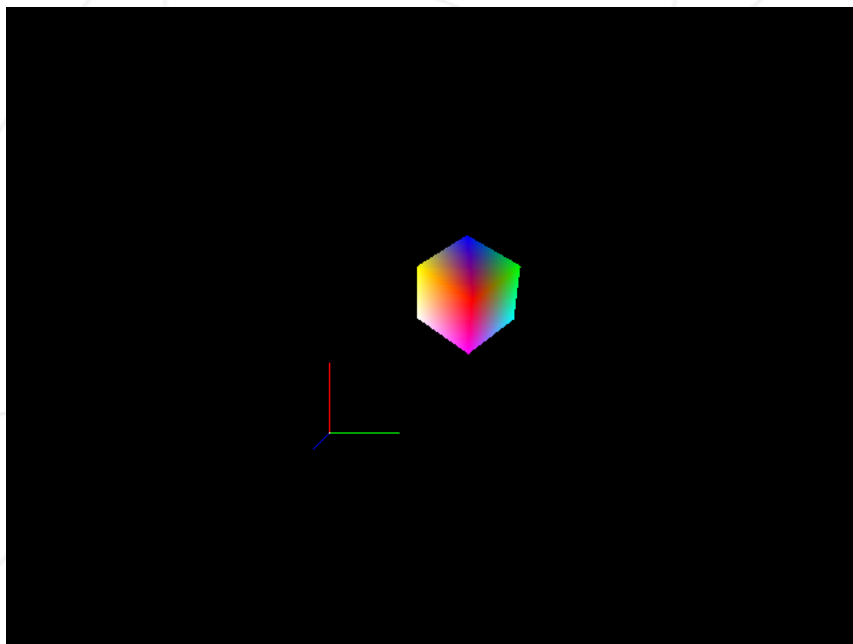
- Vertex :



- Edge :



- Raster :



L'image générée doit être au format **png**. Pour cela, toute la bibliothèque GD de PHP est à votre disposition.

L'implémentation de cet exercice va être extrêmement libre. Vous devez écrire une classe **Triangle** et une classe **Render**. Vous pouvez ajouter de nouvelles classes et/ou modifier des classes des exercices précédents. Les consignes à respecter sont les suivantes.

Concernant la classe **Triangle** :

- Le constructeur de la classe **Triangle** attend un tableau. Les clefs attendues sont les suivantes :  
  
    '**A**' : Vertex du premier point du triangle, obligatoire.  
  
    '**B**' : Vertex du second point du triangle, obligatoire.  
  
    '**C**' : Vertex du troisième point du triangle, obligatoire.
- La classe **Triangle** doit contenir un attribut **verbose statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur **False**.
- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier **Triangle.doc.txt** et est laissée à votre discrétion, comme dans les exercices précédents.
- Il n'y a pas de méthodes obligatoires pour votre classe **Triangle**. Toutefois en écrire quelques unes peut vous être utile. Je pense par exemple à pouvoir itérer ou mapper les vertices du triangle, pouvoir itérer ou mapper les arêtes du triangle, pouvoir trier les vertices ou les arêtes selon certaines conditions, etc.

Concernant la classe **Render** :

- Le constructeur de la classe **Render** attend un tableau. Les clefs attendues sont les suivantes :  
  
    '**widht**' : Largeur de l'image générée, obligatoire.  
  
    '**height**' : Hauteur de l'image générée, obligatoire.  
  
    '**filename**' : Nom du fichier dans lequel sera sauvegardée l'image png créée, obligatoire.
- La classe **Render** doit proposer trois constantes de classe **VERTEX**, **EDGE** et **RASTERIZE** qui serviront à choisir le mode de rendu.
- La classe **Render** doit contenir un attribut **verbose statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est

initialement à la valeur **False**.

- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier **Render.doc.txt** et est laissée à votre discrétion, comme dans les exercices précédents.
- votre classe **Render** doit proposer les méthodes suivantes :

**void renderVertex( Vertex \$screenVertex )** : Affiche un vertex en coordonnées "écran" dans l'image générée (un pixel quoi).

**void renderTriangle( Triangle \$triangle, \$mode )** : Affiche un triangle en coordonnées "écran" dans l'image générée selon le mode voulu. Le mode est l'une des trois constantes de classe.

**void develop()** : Sauvegarde l'image **png** générée sur le disque dur en utilisant le nom de fichier fourni au constructeur.

Pour simplifier votre travail, je vous conseille d'ajouter quelques fonctionnalités très simples à certaines des classes des exercices précédents ou à ajouter vos propres classes. Cela n'est bien sûr pas obligatoire. Voici quelques pistes sans être exhaustif :

- Une méthode **toPngColor** pour votre classe **Color** dont le pseudo code serait le suivant avec l'aide de la bibliothèque GD :


```
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
```

- Support des triangles dans vos classes **Matrix** et **Camera**.
- Ajout d'une méthode **bool isVisible( Triangle \$tri )** à la classe **Camera** pour déterminer si un triangle est visible ou non pour ne pas afficher les faces arrières d'un objet. Vous pouvez utiliser le z-buffer ou un BSP si vous le souhaitez mais il existe un algo beaucoup plus court et simple à mettre en place pour un simple exercice de piscine. Cherchez le "backface culling".
- Ajout d'une classe **Mesh** permettant de représenter un modèle 3D composé de triangles qui permettrait de manipuler facilement les vertices, les arêtes et les triangles du modèle. Bien évidemment, ajouter le support de cette classe aux classes **Matrix** et **Camera** est extrêmement pratique!
- ...

Vous trouverez un exemple d'utilisation de cette classe dans le fichier `main_05.php` en annexes de ce sujet qui a servi à générer les trois images précédentes.

# Chapitre XI

## Bonus exercice 06 : La classe Texture

	Exercice : 06
La classe Texture	
Dossier de rendu : <i>ex06/</i>	
Fichiers à rendre : <code>Texture.class.php</code> , <code>Texture.doc.txt</code> , <code>*.class.php</code> , <code>*.doc.txt</code> , vos fichiers de textures	
Fonctions Autorisées : Tout ce qui a été vu depuis le début de la piscine, toute la bibliothèque standard de PHP et la bibliothèque GD.	
Remarques : n/a	

Exercice pour les acharnés qui permet de monter au dessus de 20 points en soutenance. Ajoutez le support des textures à votre rasterizer. Vous pouvez ajouter et modifier tout ce que vous voulez.

Les consignes à respecter sont les suivantes :

- La classe **Texture** doit contenir un attribut **verbose statique** de type boolean permettant de contrôler les affichages liés à l'utilisation de la classe. Cet attribut est initialement à la valeur **False**.
- Si et seulement si l'attribut statique **verbose** est vrai, alors le constructeur et le destructeur de la classe produiront une sortie.
- La classe doit proposer une méthode statique **doc** retournant une documentation courte de la classe sous forme de chaîne de caractères. Le contenu de la documentation devra être lu depuis un fichier **Texture.doc.txt** et est laissée à votre discrétion, comme dans les exercices précédents.

Pour vous aider dans vos recherches, les notions utiles ici sont les "coordonnées **u** et **v** d'un vertex" et les coordonnées barycentriques d'un triangle ...



J'offre le café au premier qui réussit. - Thor