INTRODUCTION A LA

PROGRAMMATION WEB

1ERE ANNEE DU CYCLE INGENIEUR

TD 6

Canvas HTML5

charles.meunier@u-bourgogne.fr

OBJECTIFS

- Etudier le patron de conception Sujet / Observateur,
- Mettre en place une architecture MVC.

PARTIE 1 – PREMIER DESSIN

DE QUOI S'AGIT-IL ?

Mais avant de commencer, de quoi parlons nous exactement avec ce Canvas ? Il s'agit d'une balise HTML introduite avec HTML 5 et qui se matérialise par une surface, au sein de la page Web, sur laquelle il est possible de dessiner en 2D mais également en 3D avec l'API WebGL. Seuls les aspects 2D seront abordés ici.

MISE EN PLACE DU CANVAS

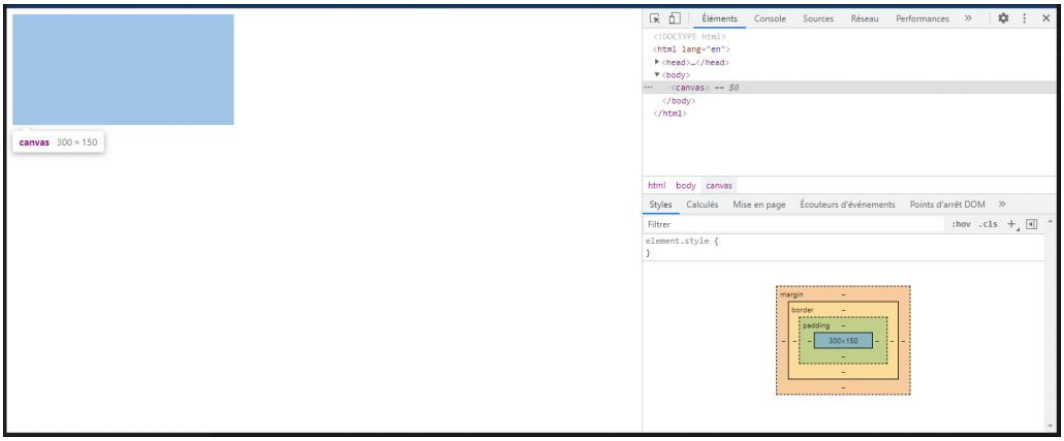
- Commencez par créer un document `index.html` avec une structure HTML de base :

```
<!DOCTYPE html>
<html>
	<head>
		<title>Canvas HTML 5</title>
	</head>

	<body>
	</body>
</html>
```

- Ajoutez une balise `canvas` dans la section `body` de la page :

```
…
<body>
	<canvas></canvas>
</body>
…
```

- Ouvrez le document `index.html` dans votre navigateur. Vous devriez… ne rien voir. Par défaut, la surface de dessin du canvas est transparente.
- Ouvrez les outils de développement intégrés à votre navigateur (Ctrl+Shit+i ou F12). Dans l'onglet « Eléments », survolez la balise canvas avec votre souris et vous devriez voir la surface apparaître.



Visualisation d'un élément de la page web à partir des outils de développement

Note : Vous pouvez constater que le canvas possède des dimensions par défaut de 300 x 150 pixels. Ceci aura sont importance pour plus tard.

DESSINE-MOI UN… CARRE ?

Vous allez réaliser votre premier dessin sur une page Web, et pour cela, quoi de mieux qu'un bon vieux carré ! C'est simple et efficace pour s'assurer que tout fonctionne bien avant d'aller plus loin.

- Créez un dossier `js` à la racine de votre site web.
- Créez, dans ce dossier, un fichier `draw.js`
- Liez ce fichier à votre document `index.html` en ajoutant une balise `script` à la fin du `body` de votre page :

```
…
<body>
	<canvas></canvas>

	<script src="js/draw.js></script>
</body>
…
```

- Editez le fichier `draw.js`
- Première chose, il va falloir récupérer l'élément HTML représenté par la balise <canvas> :

```
const canvas = document.querySelector("canvas");
```

Avoir accès à l'élément HTML n'est pas suffisant pour pouvoir dessiner. En effet, le canvas est composé de deux parties :

- l'élément HTML qui est intégré à la page et auquel vous pouvez associer un style CSS
- le contexte graphique au sein duquel sera réalisé le dessin et qui sera ensuite affiché au sein de l'élément HTML cité précédemment.

Pour pouvoir dessiner, il va donc falloir récupérer ce fameux contexte graphique :

```
const context = canvas.getContext("2d");
```

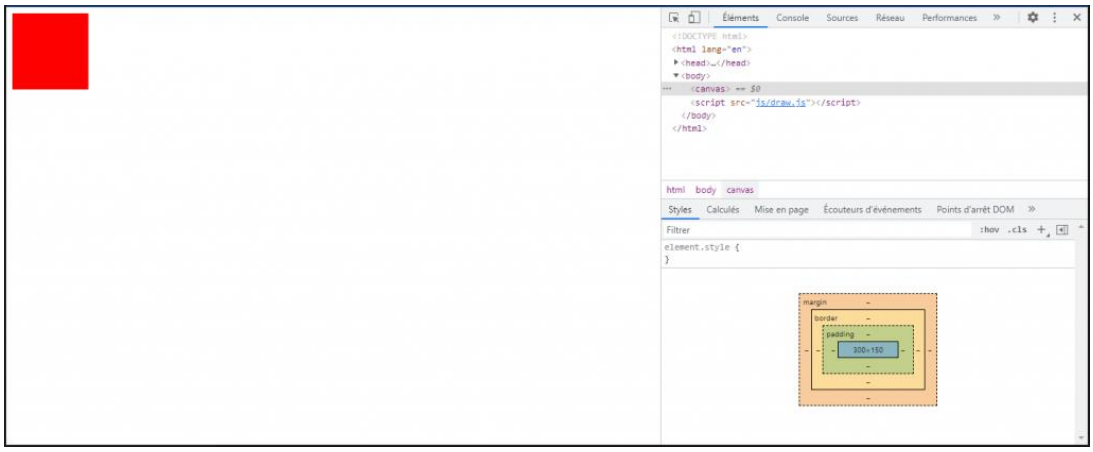
Note 1 : respectez bien la casse pour le nom du contexte « 2d » à récupérer. Si vous mettez un majuscule, cela ne fonctionnera pas.

Note 2 : Ici nous récupérons le contexte graphique permettant de réaliser des dessins en 2D. Il serait possible, comme annoncé plus haut, d'obtenir un environnement pour modéliser des scènes 3D. Nous utiliserions alors le contexte « webgl ».

- Une fois le contexte récupérer, nous allons pouvoir utiliser les primitives de dessin associées pour tracer le carré :

```
// Starts the drawing of the shape
context.beginPath();

/*
 Prepares the draweing of a rectangle starting at (0 ; 0) with 100 pixels width
and 100 pixels height
*/
context.rect(0, 0, 100, 100);

// Sets the color of filling
context.fillStyle = "#FF0000";

// Fills the square
context.fill();
```

- Enregistrez tous les fichiers et actualisez la page de votre navigateur. Vous devriez voir apparaître un carré rouge dans l'angle supérieur gauche de la page Web.



Carré rouge dessiné via le Canvas HTML 5

DIMENSIONNEMENT DU CANVAS

Vous l'avez vu précédemment, par défaut le canvas fait 300 pixels de large par 150 pixels de haut. Comme tout élément HTML, il est possible de définir la taille d'un canvas via une feuille de style.

Nous allons à présent faire en sorte que le canvas couvre la totalité de la page.

- Créez un dossier `css` à la racine de votre site et ajoutez-y un fichier `style.css`
- Associez le fichier `style.css` au document `index.html`
- Editez le fichier style.css.

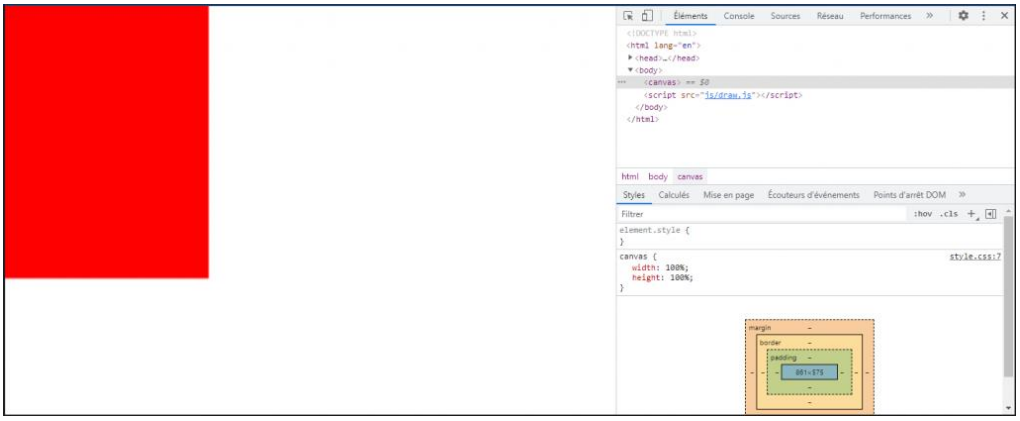
- Nous allons commencer par supprimer les marges de base du body et faire en sorte que ce dernier occupe tout l'espace disponible sur la page :

```
body {
	margin: 0;
	width: 100vw;
	height: 100vh;
}
```

- Puis nous allons indiquer au canvas d'occuper tout l'espace disponible de son parent (ici le body) :

```
canvas {
	width: 100%;
	height: 100%;
}
```

- Enregistrez tous les fichiers et actualisez la page Web. Oups ?



Conséquence d'un redimensionnement du canvas par CSS seulement

Votre carré ne ressemble plus vraiment à un carré, n'est-ce pas ? D'une part, c'est devenu un rectangle et d'autre part, ses contours sont comme flous. Et dire que tout ceci est normal. si, si, vous allez comprendre.

Si, comme plus tôt dans cet exercice, vous survolez la balise canvas dans les éléments qui constituent la page, vous verrez que la canvas couvre bien toute la surface de la page. Donc sur ce point, le CSS a fait son travail.

Mais comment cela a été dit précédemment, le canvas est constitué de deux parties : l'élément HTML et le contexte graphique. Or le fait de redimensionner l'un, n'affecte pas les dimensions de l'autre. Par ailleurs, le contexte graphique est étiré pour couvrir toute la surface de l'élément HTML.

Si l'on reprend la capture d'écran ci-dessus, l'élément HTML du canvas mesure 861 par 675 pixels, et le contexte graphique est resté sur ses dimensions initiales, à savoir 300 x 150 pixels. De manière à recouvrir l'intégralité de l'élément HTML, le contexte graphique est donc étiré environ 3 fois en largeur et 5 fois en hauteur. De fait, notre carré est devenu un rectangle.

L'étirement conduit également à une perte de résolution et à des points qui chevauchent plusieurs pixels, ce qui explique les bords flous de la forme.

Pour résoudre tous ces problèmes, il suffit d'appliquer les même dimensions à l'élément HTML et au contexte graphique.

- Ajoutez les lignes suivantes dans le fichier `draw.js`, juste avant la récupération du contexte :

```
...
canvas.width = canvas.clientWidth;
canvas.height = canvas.clientHeight;
...
```

Note : Les attributs `width` et `height` du canvas définissent la taille du contexte graphique. Les attribut `clientWidth` et `clientHeight`, communs à tous les éléments HTML, fournissent les dimensions de l'élément HTML dans la page.

- Actualisez la page. Le carré redevient carré.

Oui mais… que se passe-t-il si vous redimensionnez la fenêtre de votre navigateur ? Le rectangle s'en trouve à nouveau déformé. En effet, le redimensionnement du contexte graphique n'intervient, actuellement, qu'au début du script JS qui n'est exécuté qu'une seule fois : au chargement de la page.

Si les dimensions du canvas viennent à changer par la suite, la mise à jour des dimensions du contexte graphique ne sera pas faite.

- Créez une méthode `resize` qui actualisera les dimensions du contexte graphique du canvas :

```
function resize()
{
	canvas.width = canvas.clientWidth;
	canvas.height = canvas.clientHeight;
}
```

- Appelez la méthode `resize` juste avant la récupération du contexte.
- Ajoutez un gestionnaire d'événement `resize` à l'objet `window` qui appellera la méthode précédente :

```
window.addEventListener("resize", () => {
	resize();
});
```

- Enregistrez tous les fichiers, actualisez la page et redimensionnez la fenêtre de votre navigateur. Tada ! Le carré… a disparu !

Et oui, redimensionner le contexte graphique conduit à une réinitialisation de se dernier, et donc à un effacement des dessins précédemment réalisés. Donc, après un redimensionnement, il faut penser à retracer le dessin.

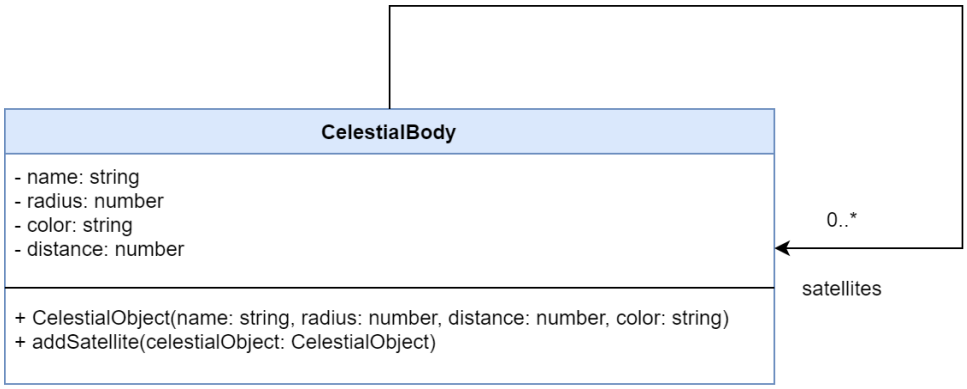
- Créez une méthode `drawSquare` qui reprendra le code permettant de dessiner le carré.
- Appelez cette méthode à la fin de la méthode `resize`

- Actualisez la page du navigateur. Cette fois-ci, le carré reste à l'écran et conserve ses dimensions lors des redimensionnements de la fenêtre du navigateur.

PARTIE 2 – REPRESENTATION DU SYSTEME SOLAIRE

MODELISATION DU SYSTEME SOLAIRE

Nous allons pouvoir passer à l'étape suivante qui consiste en la représentation graphique du système solaire. Pour cela, vous pourrez utiliser le fichier `celestial-object.js` qui contient une modélisation partielle du système solaire sous la forme d'objets célestes tournant les uns autours des autres :



Modélisation UML simplifiée de la classe CelestialObject

Étudions les attributs d'un corps céleste tel que nous les décrit le diagramme UML ci-dessus :

- Un corps céleste possède un nom (`name`) qui permettra d'identifier lequel représente le soleil, la Terre, …
- Il possède également un rayon (`radius`) qui déterminera la taille du cercle utilisé pour le représenté sur le dessin
- La couleur du cercle sera déterminé par l'attribut `color`
- Chaque corps céleste peut être le satellite d'un autre. L'attribut distance détermine alors la distance qui sépare un satellite de son « parent ».
- Téléchargez le fichier `celestial-body.js` et placez-le dans le dossier `js` de votre site :

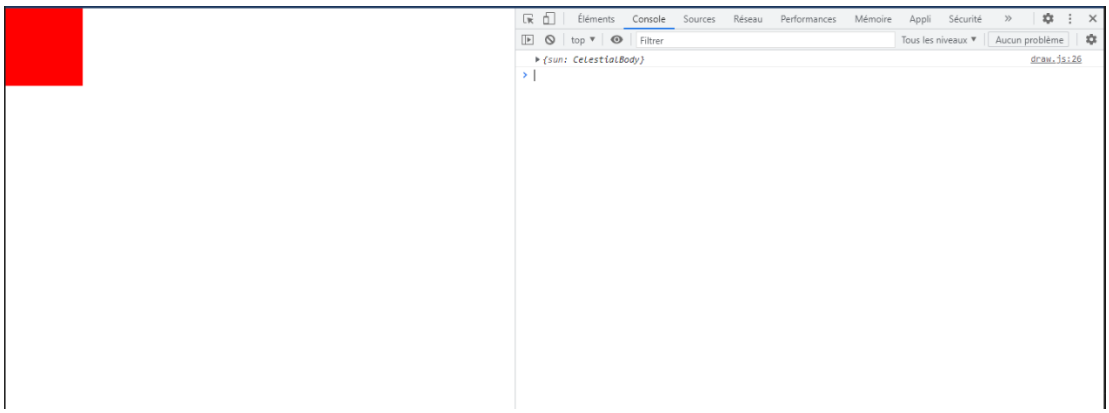
celestial-body.js

- Editez le fichier index.html et ajoutez un lien vers le script celestial-body.js en veillant à ce que ce dernier soit bien placé avant le lien vers le script draw.js :

Le fichier celestial-body.js fournit un objet `solarSystem` qui décrit partiellement le système solaire à l'aide d'objets de type `CelestialBody`.

- Editez votre fichier draw.js
- Affichez dans la console du navigateur le contenu de `solarSystem` pour bien comprendre comment se dernier est constitué.
- Enregistrez vos fichiers et actualisez la page de votre navigateur.

- Rendez-vous dans l'onglet console des outils de développement. Vous devriez obtenir quelque chose comme ceci :



- Cliquez sur la flèche située à gauche de `{ sun: CelestialBody }` afin de déployer la structure de l'objet. Vous verrez alors que `solarSystem` possède un attribut sun. Ce dernier est un `CelestialBody` qui possède 4 satellites : Mercure, Venus, la Terre et Mars. La terre possède elle-même 1 satellite : la Lune.

Vous trouverez également d'autres attributs que ceux listés dans la modélisation UML précédente. N'en tenez pas compte pour le moment, il seront utiles lors de la troisième partie de cette exercice.

DESSIN D'UN OBJET CELESTE

- Vous allez à présent créer une fonction `drawCelestialBody` qui prendra en paramètre l'objet céleste à dessiner :

```
function drawCelestialBody(celestialBody)
{
}
```

Les corps célestes seront représentés à l'aide de cercles dont la taille sera définie par l'attribut `radius`. Pour tracer un cercle, vous devrez utiliser la méthode `arc` du contexte graphique :

```
context.arc(centerX, centerY, radius, startAngle, endAngle, counterClockWise);
```

- `centerX` et `centerY` correspondent aux coordonnées du centre du cercle à dessiner
- `radius` est le rayon du cercle
- `startAngle` et `endAngle` fournissent quant à eux le point de départ et d'arrivée de l'arc de cercle à tracer, sous la forme d'angle exprimés en radians. Pour un cercle complet, il faudra donc partir de l'angle 0 et terminer à l'angle 2π (« deux fois PI »).
- Enfin, `antiClockWise` indique si le tracé doit être réalisé dans le sens trigonométrique (`true`) ou dans le sens des aiguilles d'une montre (`false`). Par défaut, ce paramètre vaut `true` .

Ainsi, dans le cadre de notre fonction drawCelestialBody, nous obtenons :

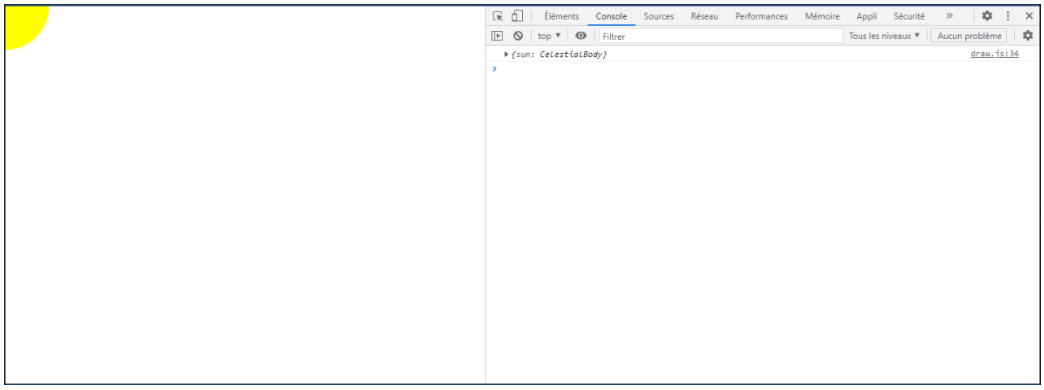
```
function drawCelestialBody(celestialBody)
{
```

```
        // Starts the drawing
        context.beginPath();

        // Prepare the drawing of a complete circle
        context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);

        // Sets the filling color
        context.fillStyle = celestialBody.color;

        // Fills the circle
        context.fill();
}
```

- Remplacez l'appel de la méthode `drawSquare` dans `resize`, par l'appel de `drawCelestialBody` en lui passant en paramètre l'attribut `sun` de `solarSystem` :

```
function resize()
{
        canvas.width = canvas.clientWidth;
    canvas.height = canvas.clientHeight;

    drawCelestialBody(solarSystem.sun);
}
```

- Enregistrez tous vos fichiers et actualisez la page Web. Vous devriez apercevoir un quart de soleil coincé dans l'angle supérieur gauche de votre page Web :



Le soleil apparaît dans l'angle supérieur gauche de l'écran

En informatique, la coordonnée (0 ; 0) est située dans l'angle supérieur gauche du support que ce dernier soit un écran, une image, ou, comme ici, un canvas.

Pour centrer le soleil dans la page, une première solution serait d'indiquer les coordonnées du centre du canvas à la méthode arc. Ainsi le centre du cercle se trouverait bien au centre de la page.

- Effectuez la modification suivante dans la méthode drawCelestialBody :

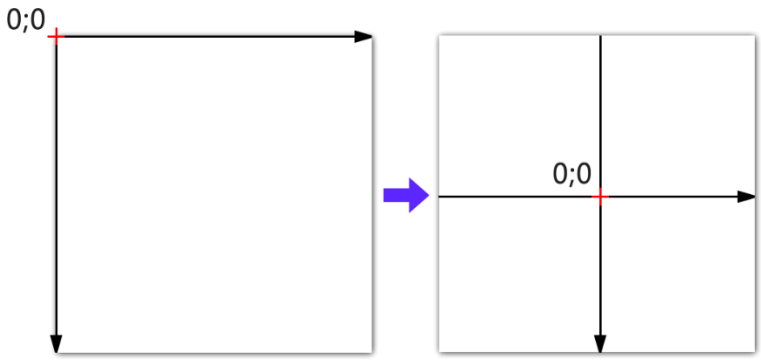
```
arc(canvas.width / 2, canvas.height / 2, celestialBody.radius, 0, 2 * Math.PI);
```

- Enregistrez la modification et actualiser la page de votre navigateur. Le soleil devrait être centré.

L'inconvénient de cette méthode est que `drawCelestialBody` détermine l'endroit où dessiner le corps céleste à partir de coordonnées absolues, ce qui va poser problème lorsque nous allons devoir dessiner les satellites.

LES TRANSFORMATIONS : TRANSLATION

Une autre solution, consiste à utiliser les transformations du canvas, qui permettent de déplacer le repère et l'emplacement de la coordonnée (0 ; 0) :



Modification du système de coordonnées

- Annulez les modifications de la méthode `drawCelestialBody` pour que le cercle ait de nouveau pour centre la coordonnée (0 ; 0).
- Créez une nouvelle méthode `drawSolarSystem` qui déplacera le repère au centre du canvas et appellera ensuite la méthode `drawCelestialBody` :

```
function drawSolarSystem()
{
	//Moves the coordinate system to the center of the canvas
	context.translate(canvas.width / 2, canvas.height / 2);

	//Draws the solar system starting with the sun
	drawCelestialBody(solarSystem.sun);
}
```

- Modifiez la méthode `resize` pour que cette dernière appelle `drawSolarSystem` au lieu de `drawCelestialBody`.
- Enregistrez et actualisez la page de votre navigateur. Le soleil se trouve toujours au centre de la page.

Attention : les transformations du système de coordonnées persistent dans le temps et se cumulent à chaque appel. Pour éviter cela, il est important de borner les transformations avec des appels aux méthodes `save` et `restore` dont les buts respectifs sont de mémoriser la configuration actuelle du contexte graphique et de la rétablir.

- Effectuez une sauvegarde du contexte graphique avant l'appel de la méthode `translate`. Puis restaurez le contexte dans son état initial à la fin de la méthode `drawSolarSystem` :

```
function drawSolarSystem()
{
	//Saves the context
```

```
	context.save();

	//Moves the coordinate system to the center of the canvas
	context.translate(canvas.width / 2, canvas.height / 2);

	//Draws the solar system starting with the sun
	drawCelestialBody(solarSystem.sun);

	//Restores the context to its states at the previous call of save
	context.restore();
}
```

RECURSIVITE

Nous sommes à présent en mesure de dessiner un corps céleste. reste à dessiner ses satellites qui sont, eux-mêmes, des corps céleste. Nous allons donc utiliser la même méthode `drawCelestialBody` pour les représenter.

- Modifiez la méthode `drawCelestialBody` pour que cette dernière dessiner le corps céleste fournit en paramètre, puis dessine ses satellites s'il en possède.

```
function drawCelestialBody(celestialBody)
{
	// Starts the drawing
	context.beginPath();

	// Prepare the drawing of a complete circle
	context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);

	// Sets the filling color
	context.fillStyle = celestialBody.color;

	// Fills the circle
	context.fill();

	// Draws each satellite of the celestial body
	celestialBody.satellites.forEach((satellite) => {
		drawCelestialBody(satellite);
	});
}
```

- Enregistrez les modifications et rechargez la page Web. Oh ! Une pokeball !

Vous en conviendrez, ce n'est pas le résultat escompté. Cela vient du fait que toutes les planètes sont dessinées aux mêmes coordonnées que le soleil. Nous allons apporter une petite correction pour régler ce problème.

Chaque corps céleste possède un attribut `distance` qui indique la distance qui le sépare de son parent. Cette donnée va nous permettre de positionner les satellites relativement par rapport à leur parent. Et, comme nous l'avons vu plus tôt, les transformations du système de coordonnées sont cumulables. Cela va rendre les choses très simples.

- Modifiez la méthode `drawCelestialBody` pour que cette dernière déplace le repère de coordonnées de la distance qui sépare le corps céleste à dessiner de son parent.

```
function drawCelestialBody(celestialBody)
{
	// Saves the context in its current state
	context.save();

	// Translates the coordinate system with the vector (celestialBody.distance ; 0)
	context.translate(celestialBody.distance, 0);

	// Starts the drawing
	context.beginPath();

	// Prepare the drawing of a complete circle
	context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);

	// Sets the filling color
	context.fillStyle = celestialBody.color;

	// Fills the circle
	context.fill();

	// Draws each satellite of the celestial body
	celestialBody.satellites.forEach((satellite) => {
		drawCelestialBody(satellite);
	});

	// Restores the context on its initial state
	context.restore();
}
```

Ca commence à ressembler à quelque chose.

- Modifiez la couleur d'arrière-plan de la page, dans le fichier `style.css`, pour que ce soit un peu plus immersif :

```
body {
	margin: 0;
	width: 100vw;
	height: 100vh;

	background-color: #1a1a1a;
}
```

Nous allons ajouter le tracé de l'orbite de chaque satellite. Pour cela nous utiliserons toujours la méthode `arc`, mais au lieu d'utiliser `fillStyle` et `fill` pour remplir le cercle, nous utiliserons `strokeStyle` et `stroke` pour tracé le contour du cercle.

- Créez une méthode `drawOrbit` qui tracera l'orbite du corps céleste fourni en paramètre :

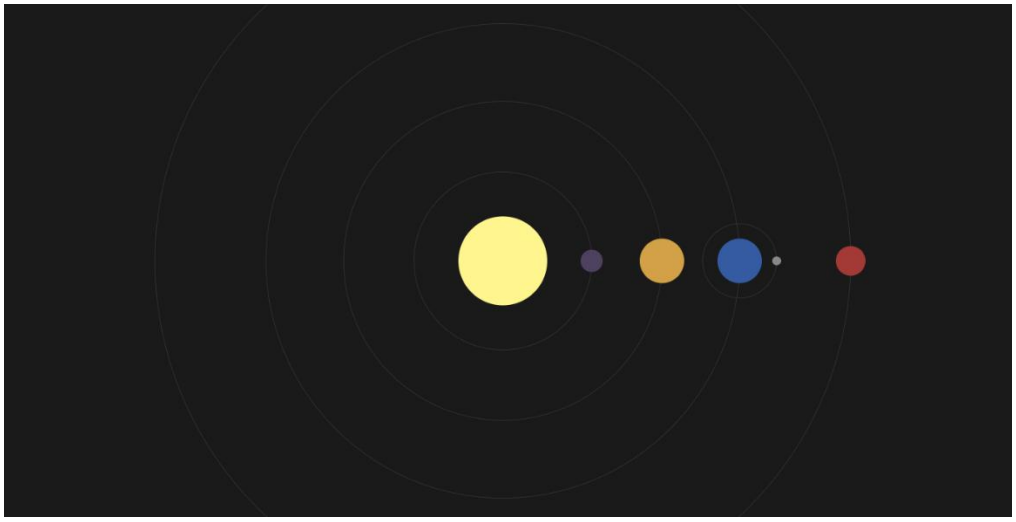
```
function drawOrbit(celestialBody)
{
	// Starts the drawing
	context.beginPath();
```

```
        // Prepare the drawing of a complete circle
        context.arc(0, 0, celestialBody.distance, 0, 2 * Math.PI);

        // Sets the outline color of the circle
        context.strokeStyle = "#333333";

        // Draws the outline of the circle
        context.stroke();
}
```

- Appelez la méthode `drawOrbit` dans la méthode `drawCelestialBody`, juste avant le dessin du satellite :

```
…
// Draws each satellite of the celestial body
celestialBody.satellites.forEach((satellite) => {
        drawOrbit(satellite);
        drawCelestialBody(satellite);
});
…
```

Vous devriez obtenir un système solaire partiel comme ci-dessous :



Représentation statique du système solaire

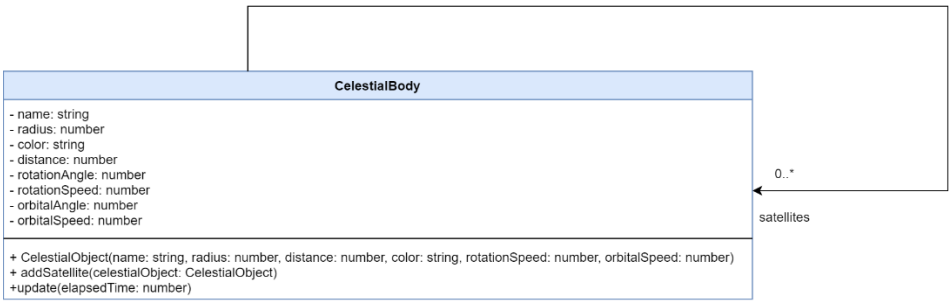
Voyons à présent comment animer tout cela !

PARTIE 3 – ANIMATION

Une animation, comme un film, c'est avant tout une série d'images légèrement différentes qui affichées l'une après l'autre vont donner l'illusion d'un mouvement.

Animer notre système solaire va donc consister à le dessiner, puis faire évoluer la position des planètes, dessiner à nouveau, mettre à jour la position des planètes, …

Pour la mise à jour de la position des planètes, tout est prêt. Vous vous souvenez, un peu plus tôt, il vous était demandé d'ignorer certains attributs de CelestialBody. Il vont prendre tout leur sens dès à présent :



Modélisation UML de la classe CelestialBody complétée

Parmi les nouveaux attributs, on retrouve :

- `rotationAngle` et `rotationSpeed` qui décrivent respectivement l'angle et la vitesse de rotation de l'astre sur lui-même.
- `orbitalAngle` et `orbitalSpeed` représentent quant à eux l'angle et la vitesse de rotation du satellite autour de son parent.
- La méthode `update` qui met à jour les positions angulaires des objets célestes en fonction du temps écoulé. Cette méthode est récursive et met à jour la position des satellites de l'objet.

ANIMATION PAR INTERVALLE

Comme vu précédemment, pour animer le dessin du système solaire, il va falloir régulièrement appeler la méthode `drawSolarSystem` puis mettre à jour la position des planètes en appelant la méthode `update`.

`setInterval` crée un déclencheur qui exécute un traitement à intervalle régulier.

- Dans le fichier `draw.js`, créez une méthode `animate` qui, toutes les 50ms, appelle la méthode `drawSolarSystem` suivi de la méthode `update` de `solarSystem.sun` :

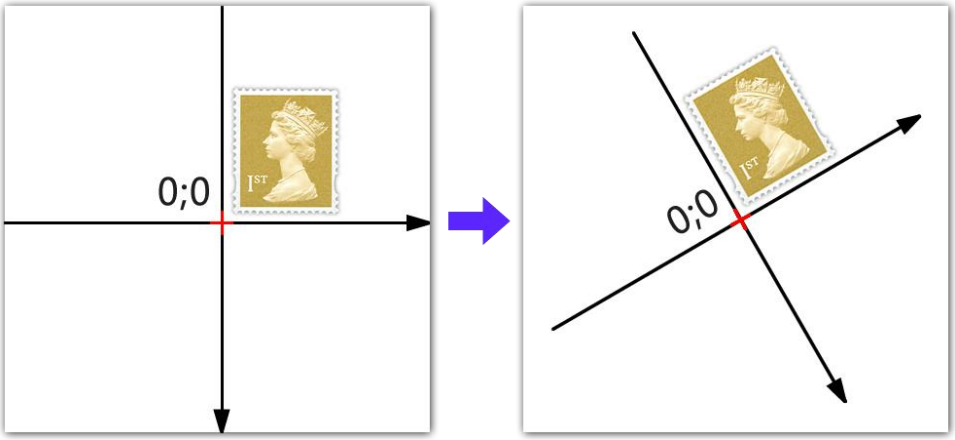
```
function animate()
{
	// Executes the callback each 50ms
	setInterval(() => {
		// Draws the solar system
		drawSolarSystem();

		// Updates celestial bodies position
		solarSystem.sun.update(50);
	}, 50);
}
```

Pour rappel, la méthode `update` de `CelestialBody` est récursive. Appeler la méthode `update` de `solarSystem.sun`, mettra à jour la position du soleil, puis de ses satellites, des satellites de ses satellites, et ainsi de suite.

L'animation est prête, reste à prendre en compte la rotation des satellites lors du dessin de ces derniers.

LES TRANSFORMATIONS : ROTATION

Nous avons vu précédemment comment déplacer le système de coordonnées pour dessine des objets les uns par rapport aux autres. Il est également possible de faire de même avec des rotations du repère. De cette façon, nous nous économisons des calculs à base de cosinus et de sinus pour faire tourner un objet autour d'un autre. Et surtout, il devient possible de faire pivoter des objets complexes comme une image. En effet, les primitives de dessin pour les images ne permettent de représenter ces dernières qu'horizontalement. Comment, dans ces conditions, la faire pivoter ? En pivotant le repère du canvas !

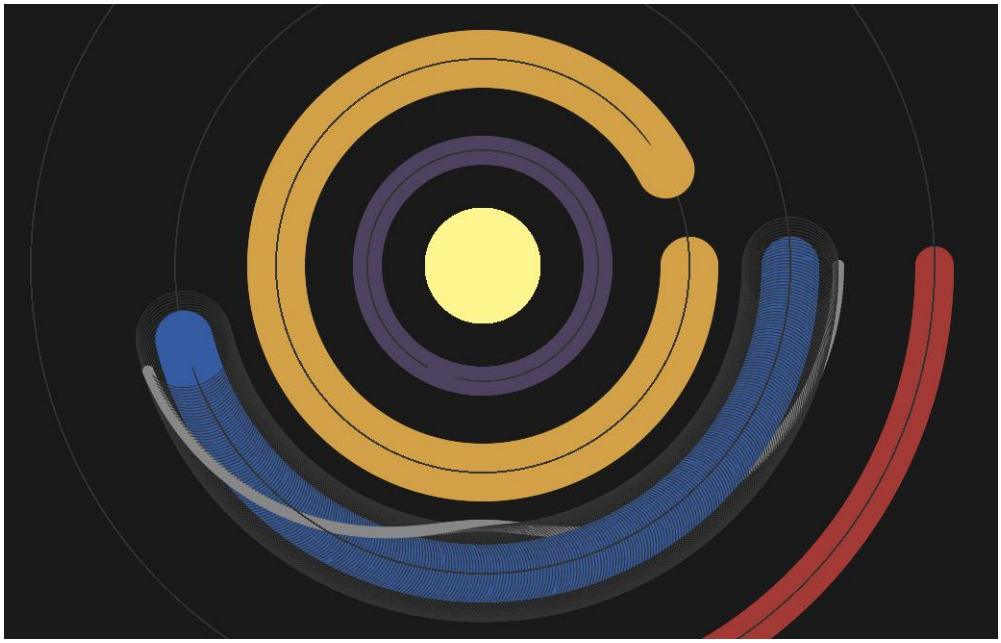


Rotation du repère

- Nous allons procéder de même pour faire tourner les satellites autour de leur parent.
- Modifiez la méthode `drawCelestialBody` pour faire pivoter le repère avant de réaliser la translation :

```
function drawCelestialBody(celestialBody)
{
	// Saves the context in its current state
	context.save();

	// Rotates the coordinate system
	context.rotate(celestialBody.orbitalAngle);

	// Translates the coordinate system with the vector (celestialBody.distance ; 0)
	context.translate(celestialBody.distance, 0);

	...
}
```

- Enregistrez les modifications et actualisez la page de votre navigateur :



Ok, ça tourne, mais il manque quelque chose : effacer le canvas entre deux dessins. Pour cela, nous allons utiliser la méthode `clearRect` du contexte graphique :

```
context.clearRect(x, y, width, height);
```

Cette méthode efface la zone définie par le rectangle de largeur `width`, de hauteur `height` et ayant son angle supérieur gauche aux coordonnées (`x` ; `y`).

- Modifiez la méthode `animate` afin d'effacer le canvas entre deux dessins :

```
function animate()
{
	// Executes the callback each 50ms
	setInterval(() => {
		// Clears the canvas
		context.clearRect(0, 0, canvas.width, canvas.height);

		// Draws the solar system
		drawSolarSystem();

		// Updates celestial bodies position
		solarSystem.sun.update(50);
	}, 50);
}
```

- Enregistrez les modifications et actualisez la page. Ce devrait-être beaucoup mieux.

ANIMATION PAR REQUESTANIMATIONFRAME

L'animation actuelle fonctionne à partir d'un dessin réaliser à intervalle régulier. Ceci peut poser problème car nous ne savons pas combien de temps sera nécessaire à la réalisation du dit dessin. Le risque, lorsque l'on veut animer une scène avec un grand nombre d'images par seconde, est d'avoir un temps de rendu plus long que l'intervalle fixé entre deux images. Et là, c'est le drame.

Pour pallier à ce problème, la méthode `requestAnimationFrame` permet d'effectuer un traitement aussi souvent que possible tout en garantissant que le traitement précédent est terminé avant de lancer le suivant.

L'inconvénient sera qu'il faudra déterminer le temps qui s'est écoulé depuis le dernier dessin pour pouvoir mettre à jour convenablement la position des planètes. Pour cela, nous utiliserons `performance.now()` qui retourne le nombre de millisecondes écoulées depuis l'ouverture de la page Web.

- Modifiez la méthode `animate` pour utiliser `requestAnimationFrame` au lieu de `setInterval` et calculez le temps écoulé entre chaque animation pour mettre à jour convenablement la position des planètes :

```
function animate(lastUpdateTime)
{
	// Gets the number of milliseconds elapsed from the beginning of the program
	const now = performance.now();

	// Computes the elpased time from the last update.
	// If lastUpdateTime is equel to 0, it is the first frame, so update is not required.
	const elapsedTime = lastUpdateTime === 0 ? 0 : now - lastUpdateTime;

	// Clears the canvas
	context.clearRect(0, 0, canvas.width, canvas.height);

	// Draws the solar system
	drawSolarSystem();

	// Updates celestial bodies position
	solarSystem.sun.update(elapsedTime);

	// Requests a new frame as soon as possible
	requestAnimationFrame(() => { animate(now) });
}
```

- Enregistrez, actualisez et admirez la fluidité !

PARTIE 4 – PATTERNS, MASQUES ET DEGRADES

CHARGEMENT DES TEXTURES

Si la représentation du système solaire, dans cet exercice, n'a pas vocation à être fidèle, tout cela reste très schématique. Voyons si nous ne pouvons pas faire mieux en intégrant quelques textures.

Pour cela, nous allons utiliser les `Patterns` que fournit le canvas pour texturer des formes. Mais avant cela, il nous faut des textures !

- Créez un dossier `img` à la racine de votre site.
- Télécharger les fichiers suivants et placez-les dans le dossier `img` précédemment créé :

sun.png / earth.png / mercury.png / moon.png / venus.png / mars.png

Il est important de bien conserver le nom des images téléchargées pour la suite de l'exercice.

Le processus de chargement des textures est déjà intégré à `CelestialBody`. Toutefois, il va être nécessaire de déclencher celui-ci lorsque votre page Web sera chargée.

- Dans `draw.js`, ajoutez un gestionnaire d'événements `load` à l'objet `window` qui initialisera les textures des corps célestes puis redimensionnera le canvas et démarrera l'animation :

```
window.addEventListener("load", () => {
	solarSystem.sun.initTexture().then(() => {
		resize();
		animate();
	});
});
```

`initTexture` est une méthode asynchrone qui retourne une promesse. Deux notations sont possibles : celle indiquée précédemment et celle ci-dessous qui a ma préférence (n'oubliez pas le `async`) :

```
window.addEventListener("load", async () => {
	await solarSystem.sun.initTexture()
	resize();
	animate();
});
```

`initTexture` est récursive et initialise la texture du corps céleste et de ses éventuels satellites. Un seul appel pour le soleil est donc nécessaire pour initialiser toutes les textures du systèmes.

Important : les méthodes `resize` et `animate` seront appelées sitôt que la page aura terminée de charger tous ses éléments. Il n'est donc plus nécessaire d'appeler ces méthodes à la fin du script. Pensez bien à retirer les retirer.

- Pour vérifier que les textures sont bien chargées, ajoutez la ligne suivante à la fin du gestionnaire d'événements précédent :

```
console.log(solarSystem.sun.texture);
```

- Enregistrez vos fichiers, actualisez la page Web et vérifiez dans la console de développement que vous n'avez pas d'erreur et que vous obtenez un affichage similaire à celui-ci :

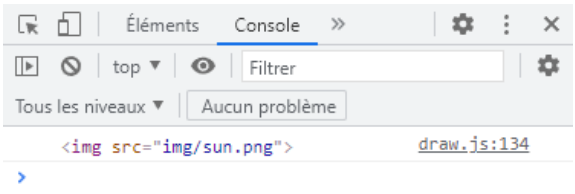


Image chargée pour texturer le soleil

- Vous pouvez retirer le `console.log` précédent.

LES PATTERNS

Les textures étant chargée, il va falloir maintenant les appliquer aux disques qui représente le soleil et les planètes du système solaire.

Actuellement, nous utilisons une couleur unie pour définir le style de remplissage des formes :

```
context.fillStyle = "#FF0000";
```

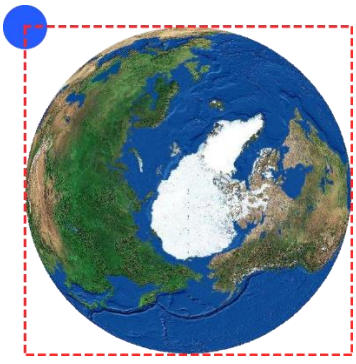
Pour appliquer une texture comme style de remplissage, nous allons devoir créer un motif (Pattern) à partir de cette image, et c'est ce motif qui sera ensuite appliqué à la forme.

- Modifiez la section de code de `drawCelestialBody` qui définit la couleur de remplissage des cercles :

```
context.beginPath();
context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);

const pattern = context.createPattern(celestialBody.texture, "no-repeat");
context.fillStyle = pattern;

context.fill();
```

- Enregistrez et actualisez la page.

On dirait bien que les planètes ont disparues ! Cela vient du fait que les textures utilisées sont plus grandes que les disques dessinés. Et comme les textures représentent des objets ronds, nous passons à coté :



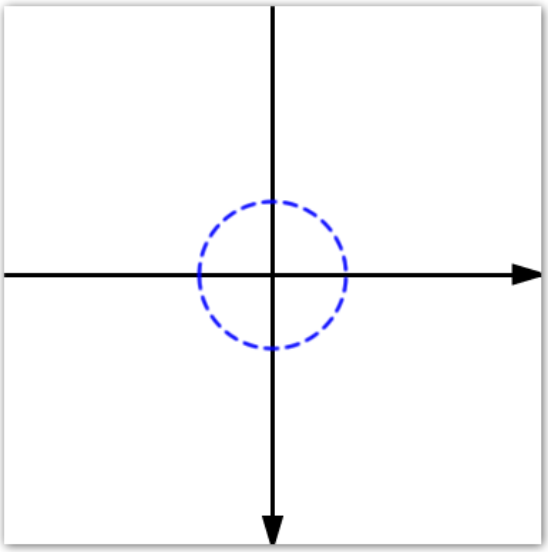
Positionnement du pattern par rapport au disque bleu

Comme vous pouvez le voir sur l'illustration ci-dessus, la pattern n'est pas automatiquement redimensionné pour couvrir l'objet. Il va donc falloir traiter cela nous-mêmes et, là encore, les transformations vont nous être très utiles.

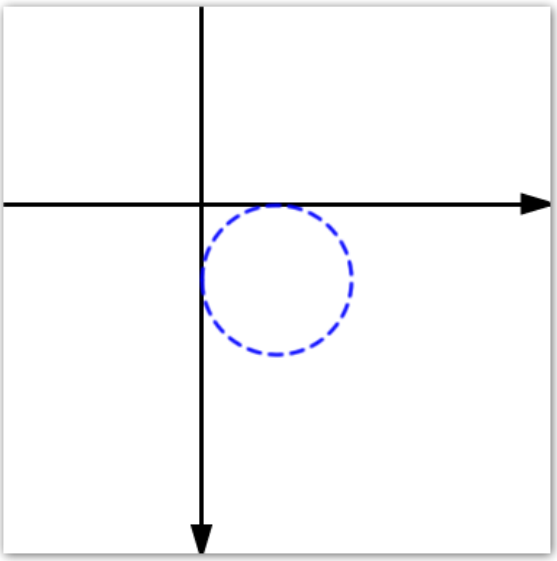
Première chose à faire : définir le coefficient d'échelle à appliquer à la texture pour qu'elle fasse la même taille que le disque à remplir :

```
const coefEchelle = (celestialBody.radius * 2) / celestialBody.texture.width;
```

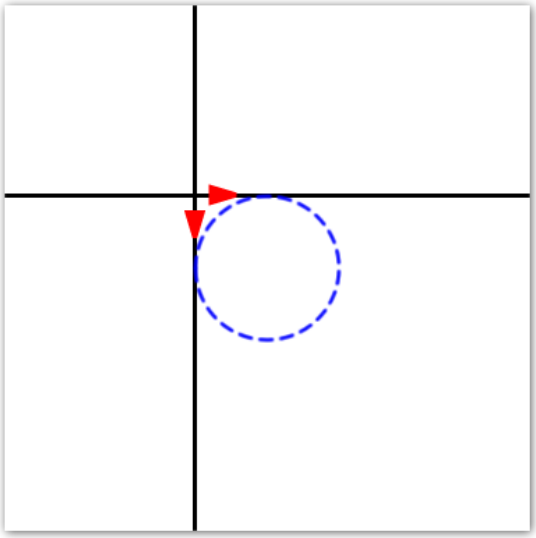
Malheureusement, nous ne pouvons pas directement le coefficient d'échelle au pattern. Il va falloir l'appliquer temporairement au contexte graphique, appliquer le pattern puis revenir à l'échelle initiale. Cela peut sembler étrange, mais une fois que l'on a compris le truc, tout s'éclaire :



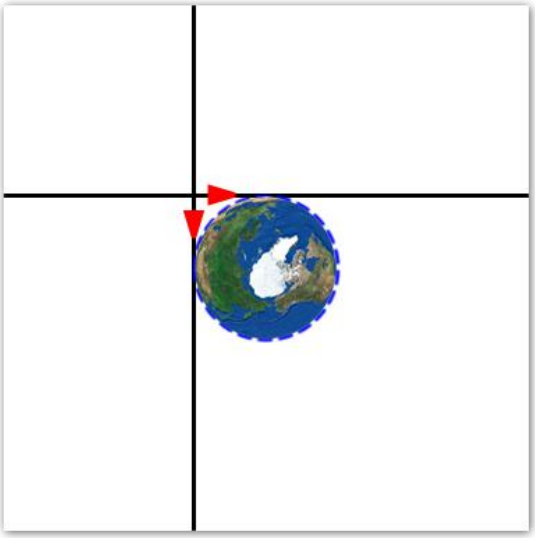
1. On prépare le tracé du cercle



2. On déplace le repère dans « l'angle » supérieur gauche du disque



3. On réduit l'échelle du repère (tout ce qui sera tracé maintenant sera plus petit)



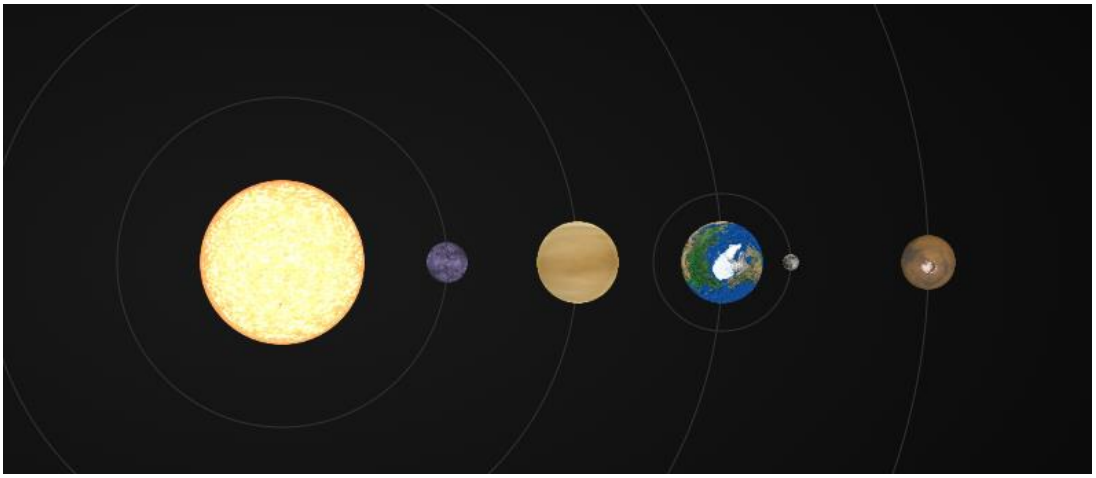
4. On remplit le cercle avec le pattern qui apparaîtra avec l'échelle en cours.

Voyons ce que cela donne dans notre cas :

```
// Prepares the drawing of the circle
context.beginPath();
context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);

// Creates the pattern and sets the fill style with it
const pattern = context.createPattern(celestialBody.texture, "no-repeat");
context.fillStyle = pattern;

// Computes the scale required to apply the pattern at the good dimensions
const coefEchelle = (celestialBody.radius * 2) / celestialBody.texture.width;
```

```
// Saves the current context
context.save()

// Translates the coordinate system to the top right corner of the circle
context.translate(-celestialBody.radius, 0);

// Sets the scales of the coordinate system horizontally and vertically
context.scale(coefEchelle, coefEchelle);

// Fills the previously prepared circle with the scaled pattern
context.fill();

// Restores the coordinate system to its initial state
context.restore();
```

- Modifiez le code de la méthode `drawCelestialBody` à partir des informations précédentes.
- Enregistrez les modifications et actualisez la page web. Tada !



ROTATION DES PLANETES

Dans la partie précédente, nous avions mis en place la rotation orbitale des planètes. Mais nous ne nous étions pas occupés de leur rotation sur leur propre axe. Si cela ne se voyait pas avec des disques de couleur unie, avec des textures, ce n'est plus possible !

La classe `CelestialBody` possède un attribut `rotationAngle` qui détermine l'angle de rotation de la planète sur elle-même.

- Modifiez la méthode `drawCelestialBody` afin d'intégrer cette rotation supplémentaire.

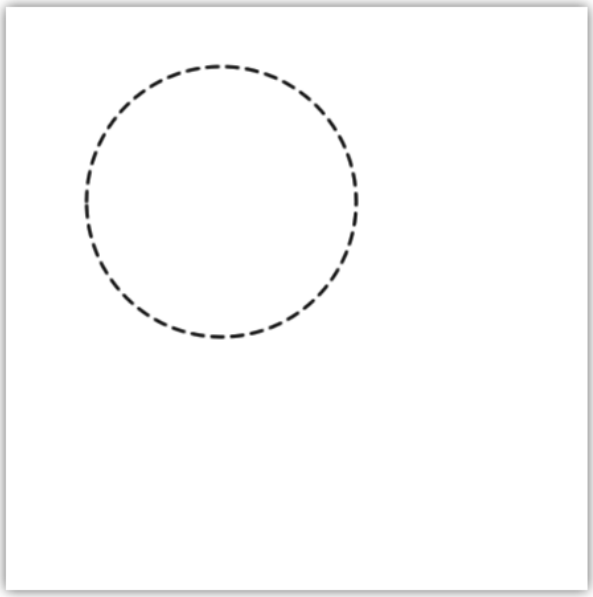
MASQUES

Parfait, notre système solaire est animé de manière cohérente. Il manque encore une petite chose, non des moindres, qui apportera une touche de réalisme incroyable : les ombres.

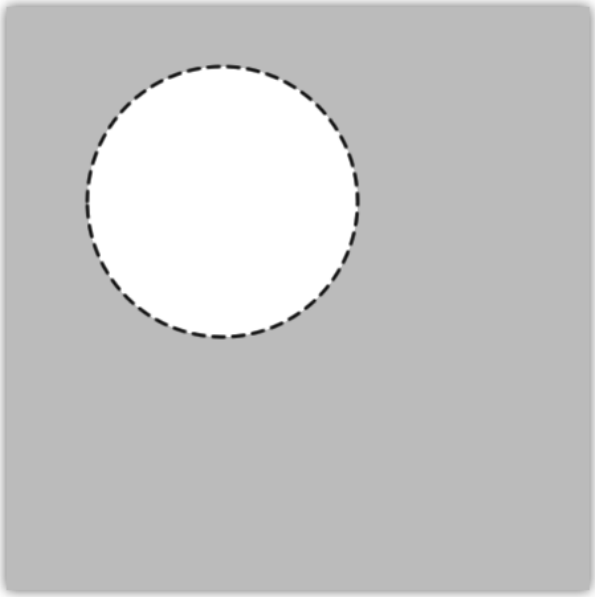
En effet, nos planètes tournent autour du soleil mais elles ne sont pas affectées par la lumière qui leur parvient. Avec une bibliothèque 3D comme Three.js ou Babylon.js, ce serait très simple à mettre en place. Mais, ici, nous allons gérer tout cela en 2D et manuellement.

Pour cela, nous allons utiliser une fonctionnalité très puissante du Canvas HTML5 : les masques.

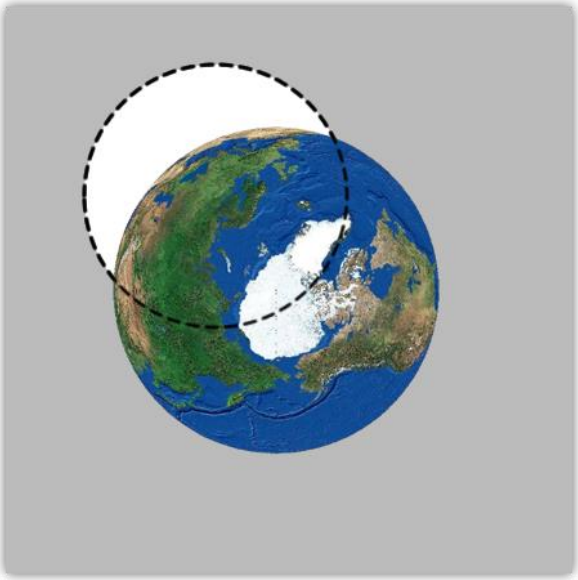
Le principe est simple : nous traçons le contour d'une forme puis nous appelons la méthode clip du contexte graphique. A partir de là, seuls les tracés contenus à l'intérieur de la forme précédente apparaîtront sur le dessin final :



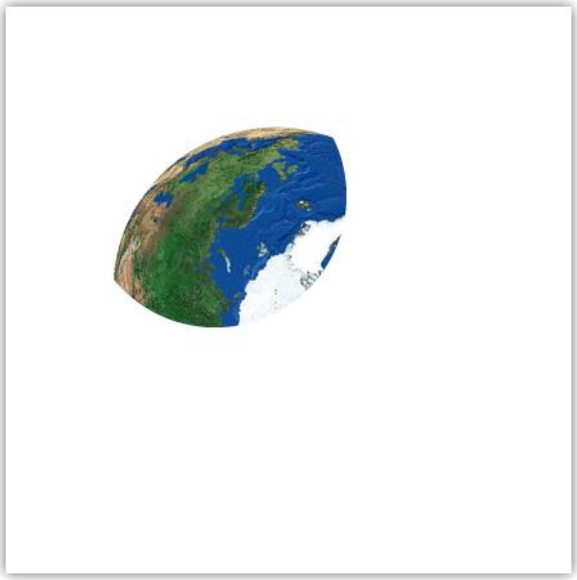
1. Sauvegarde du contexte (`save`) et tracé du contour du masque



2. Activation du masque (`clip`).

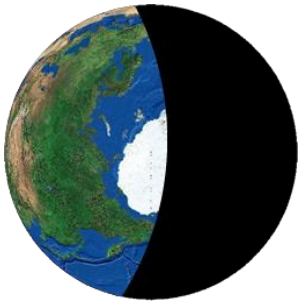


3. Tracé des éléments qui seront soumis au masque.



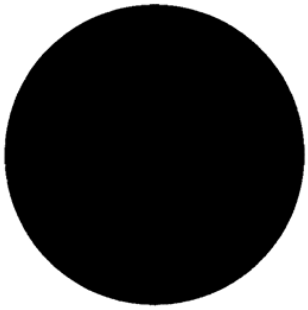
4. Restauration du contexte pour mettre fin au masquage (`restore`).

Ici, nous allons utiliser les masques pour donner l'impression qu'une face de la planète est éclairée et une autre plongée dans l'obscurité :

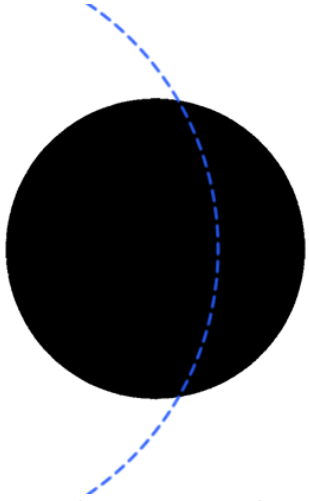


Simulation d'ombre avec un masque

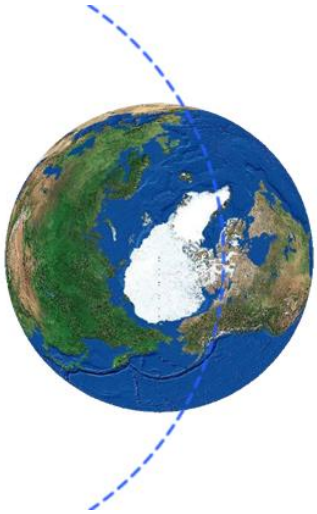
Pour obtenir ce résultat, nous allons précéder en quatre étapes :



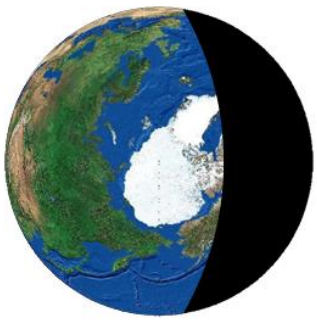
1. Dessinez un disque noir de la taille de la planète qui jouera le rôle de la face à l'ombre du soleil.



2. Sauvegardez le contexte, préparez (sans le tracé) le contour d'un cercle, au moins deux fois plus grand que la planète et décalé sur la gauche, et activer le masque (`clip`)

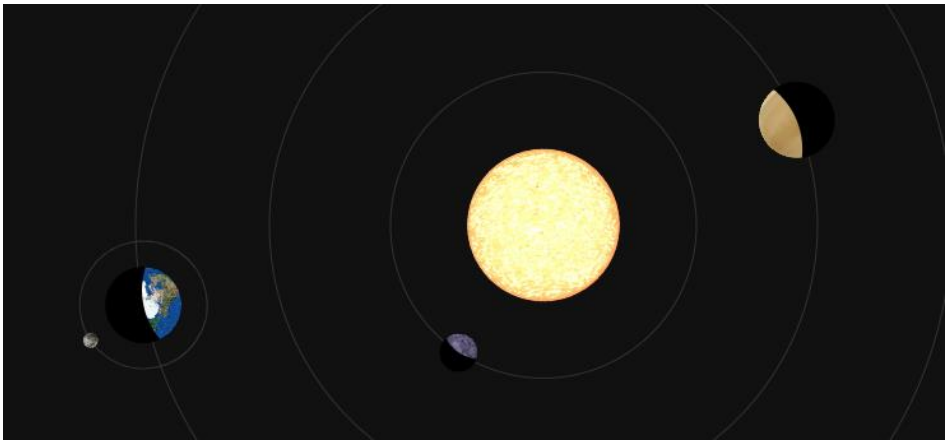


3. Dessinez un disque avec la texture de la planète qui recouvre le disque noir de l'étape 1.



4. Restaurez le contexte (`restore`) avant de poursuivre le dessin.

- Modifiez la méthode `drawCelestialBody` pour appliquer une ombre aux corps célestes qui le nécessitent (`CelestialBody` possède l'attribut `hasShadow` qui est à `true` si la planète projette une ombre).
- Enregistrez et testez. Vous devriez obtenir quelque chose comme ceci :



Planètes avec face éclairée

Niveau code, voici ce que vous devriez avoir :

```
function drawCelestialBody(celestialBody)
{
	//Saves the context in its current state [*0]
	context.save();

	context.rotate(celestialBody.orbitalAngle);

	//Translates the coordinate system with the vector (celestialBody.distance ; 0)
	context.translate(celestialBody.distance, 0);

	if(celestialBody.hasShadow)
	{
		//Draws a black disque which will be the shadowed part of the planet
		context.beginPath();
		context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);
		context.fillStyle = "#000000";
		context.fill();

		//Saves the current context [*1]
		context.save();

		//Prepares the drawing of the mask
		context.beginPath();
		context.arc(-celestialBody.radius * 2, 0, celestialBody.radius * 2, 0, 2 *
Math.PI);

		//Create a mask from the previous prepared drawing
		context.clip();
	}

	//Starts the drawing
	context.beginPath();

	//Prepare the drawing of a complete circle
	context.arc(0, 0, celestialBody.radius, 0, 2 * Math.PI);

	//Creates a pattern from the texture of the celestial body
	const pattern = context.createPattern(celestialBody.texture, "no-repeat");
	const coef = (celestialBody.radius * 2) / celestialBody.texture.width;
```

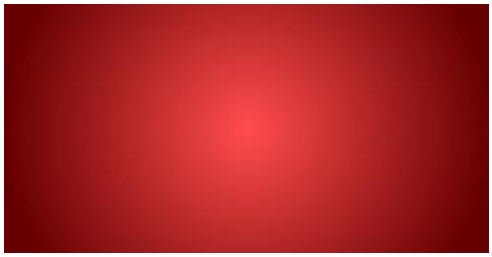
```
	//Saves the current context [*2]
	context.save();

	//Rotates the celestial body on its own axis
	context.rotate(celestialBody.rotationAngle);

	//Moves and scales the coordinate system to apply the pattern
	context.translate(-celestialBody.radius, -celestialBody.radius);
	context.scale(coef, coef);

	//Sets the filling color
	context.fillStyle = pattern; //celestialBody.color;

	//Fills the circle
	context.fill();

	//Restores the context [*2]
	context.restore();

	if(celestialBody.hasShadow)
	{
		//Restores the context and disable the mask [*1]
		context.restore();
	}

	//Draws each satellite of the celestial body
	celestialBody.satellites.forEach((satellite) => {
		drawOrbit(satellite);
		drawCelestialBody(satellite);
	});

	//Restores the context on its initial state [*0]
	context.restore();
}
```

Nous allons ajouter un dernier effet qui permettra de rendre moins net la séparation entre la face éclairée et la partie à l'ombre des planètes.

DEGRADES

Le canvas permet de réaliser des dégradés linéaires et radiaux :

Dégradé linéaire



Dégradé radial

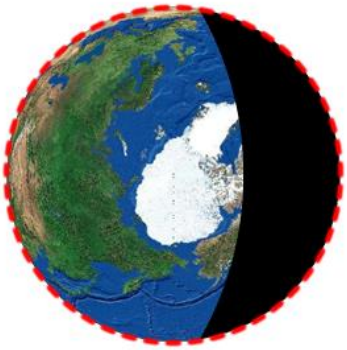
Dans notre cas, nous allons utiliser un dégradé radial pour appliquer un effet d'ombre progressive sur la bordure de la zone éclairée de la planète :

Planète avec une ombre plus progressive

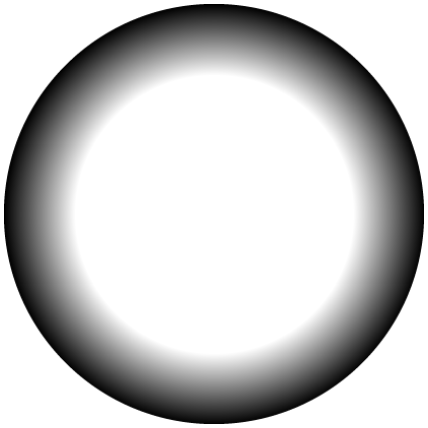
La création d'un dégradé se fait en deux temps :

1. On commence par déterminer les coordonnées de début et de fin du dégradé (plus les rayons dans les cas d'un dégradé radial) : `context.createLinearGradient` ou `context.createRadialGradient`
2. On définit les teintes par lesquelles passera le dégradé : `gradient.addColorStop`

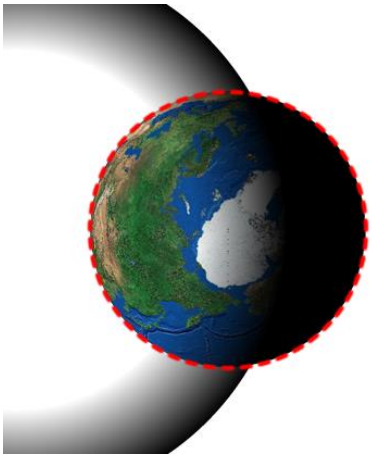
Pour obtenir le résultat ci-dessus, nous allons cumuler masque et dégradé radial. Nous allons procéder en quatre étapes :



1. Enregistrez le contexte actuel, préparé le dessin d'un cercle de la taille de la planète et activer le masque.



2. Créez un dégradé radial transparent au centre et noir sur le pourtour. Les coordonnées du dégradé devront correspondre à celle du masque qui a permis de créer la face éclairée de la planète

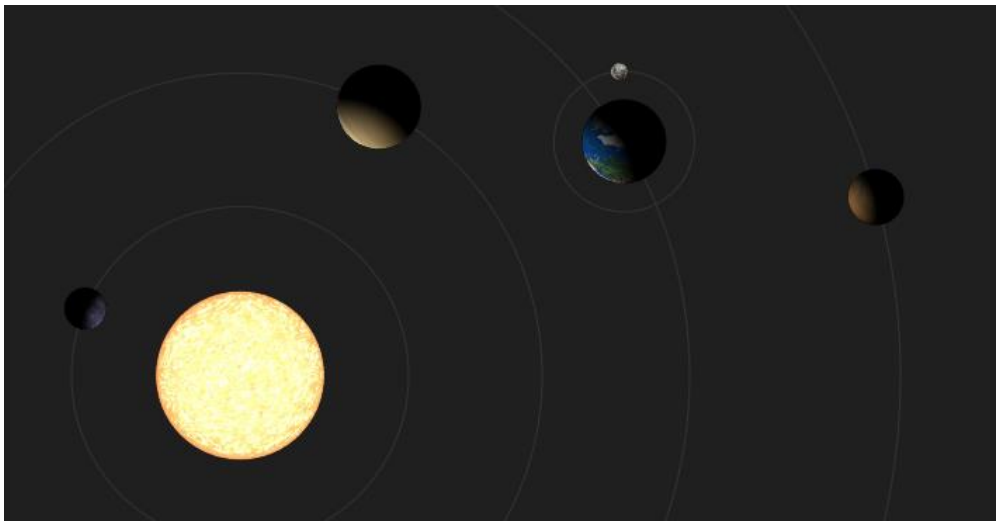


3. Remplir un cercle (identique à celui qui a permis de réaliser la face éclairée de la planète) avec le dégradé précédemment créé.

4. Restaurez le contexte afin de désactiver le masque.

- Modifiez la méthode `drawCelestialBody` pour les planètes qui possède une ombre.
- Enregistrez votre fichier et actualisez votre page Web.

Votre système devrait ressembler à cela :



Planètes avec des ombres progressives

Pour terminer et ajouter la touche ultime, nous allons appliquer un dégradé CSS au niveau de l'arrière-plan. En effet, nous avons affecter une teinte unie plutôt dans l'exercice. Or, avec un dégradé qui s'assombrit au fur et à mesure que l'on s'éloigne du soleil, le résultat sera top !

- Dans le fichier style.css, remplacez la propriété **`background-color`** du body par la ligne suivante :

```
background: radial-gradient(#1a1a1a, #000000);
```

- Enregistrez tous vos fichiers une dernière fois et admirez le travail !

Félicitations ! Vous êtes parvenu à la fin de cet exercice. Vous avez parcouru les principales fonctionnalités du Canvas HTML 5. Il en reste d'autres comme les filtres et les compositions qui permettent d'aller encore plus loin.