40-Dessiner en OpenGL en utilisant jit.gl.sketch

Les capacités OpenGL de Jitter nous fournissent une variété d'outils pour créer des scènes animées en trois dimensions. Les objets OpenGL de Jitter que nous avons examinés jusqu'ici effectuent chacun une tâche assez simple, qu'il s'agisse de créer des géométries simples (*jit.gl.gridshape* et *jit.gl.plato*), d'importer des modèles OBJ (*jit.gl.model*) ou de manipuler la position des objets et des scènes (*jit.gl.handle*). Des scènes plus complexes peuvent être construites en utilisant plusieurs de ces objets à la fois. Cependant, à partir d'un certain point, il peut être utile de travailler avec un objet qui comprend directement les commandes OpenGL et peut être utilisé pour exécuter un grand nombre de commandes de dessin à la fois. Cet objet est *jit.gl.sketch*, et c'est le sujet de ce tutoriel.

L'objet *jit.gl.sketch* s'interface avec le système OpenGL de Jitter tout comme les autres objets OpenGL; par conséquent, il peut être utile de revoir le *didacticiel 30*: ***Dessin de texte 3D*** et le *didacticiel 31*: ***Destinations de rendu avant de commencer***. De plus, les messages et les attributs compris par *jit.gl.sketch* ont une forte ressemblance avec les méthodes et propriétés de l'objet JavaScript sketch utilisé dans l'objet Max *jsui*. La section *Conception d'interfaces utilisateur* vous fournira des informations supplémentaires sur les capacités des graphiques vectoriels OpenGL qui sont communes à ces deux systèmes.

La majorité des messages utilisés par *jit.gl.sketch* sont des versions légèrement modifiées des fonctions standard de l'API OpenGL. L'ensemble de l'API sort du cadre de ce didacticiel; toutefois, la référence standard de ces fonctions (le «Redbook» d'OpenGL) est disponible en ligne à l'adresse suivante:
http://www.opengl.org/sdk/docs/man4/

La conversion entre le code OpenGL tel qu'il est donné dans le «Redbook» et ailleurs (dans le langage de programmation C) et les messages vers *jit.gl.sketch* est raisonnablement simple si l'on garde à l'esprit les conventions suivantes:

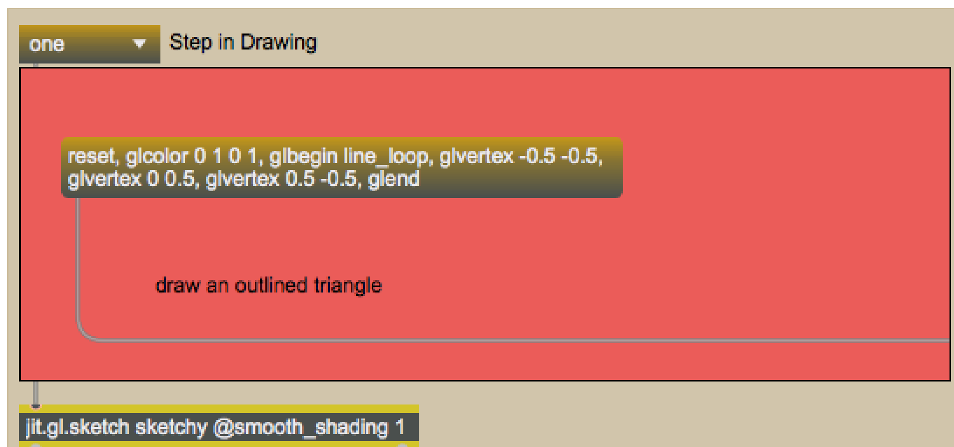
• Toutes les commandes OpenGL sont en minuscules dans *jit.gl.sketch*. Ainsi, la fonction OpenGL **glColor ()** devient le message Max **glcolor** envoyé à *jit.gl.sketch*.

• Les constantes symboliques OpenGL, en plus d'être en minuscules, perdent leur préfixe «GL_», de sorte que *GL_CLIP_PLANE1* (dans OpenGL) devient ***clip_plane1*** lorsqu'il est utilisé avec *jit.gl.sketch*, par exemple.

En plus de l'API OpenGL de base, *jit.gl.sketch* comprend également un certain nombre de commandes de dessin de haut niveau pour demander à l'objet de rendre des formes de base et d'effectuer des opérations de dessin de style graphique vectoriel.

• Ouvrez le patch du didacticiel.

Le patch du didacticiel consiste en une série d'objets Jitter utilisés pour le rendu dans OpenGL: un objet *jit.gl.render*, un objet *jit.pwindow* et un objet *jit.gl.sketch*. L'objet *jit.gl.sketch* reçoit des messages d'un objet *bpatcher* contenant des objets de boîte de *message* contenant des listes de commandes. Différentes vues du *bpatcher* peuvent être obtenues en sélectionnant différents offsets de l'objet *umenu* qui lui est connecté. Dans ce didacticiel, nous allons parcourir une à une les différentes sections du *bpatcher* et examiner les messages contenus dans chaque vue.



Notre objet *jit.gl.sketch* recevant des messages depuis un *bpatcher*.

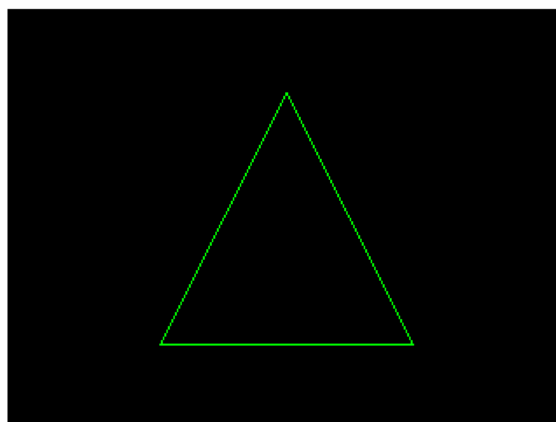
• Cliquez sur la boîte *toggle* intitulée ***Display*** attachée à l'objet *qmetro* en haut à gauche du patch.

En démarrant l'objet *qmetro*, nous avons commencé le rendu de notre scène OpenGL. Notre objet *jit.gl.render* partage son **name** (***sketchy***) avec l'objet *jit.pwindow* et l'objet *jit.gl.sketch* dans le patch. De fait, la *jit.pwindow* affichera tout ce qui est dessiné par l'objet *jit.gl.render*, qui à son tour dessinera tout ce que l'objet *jit.gl.sketch* lui demandera de dessiner.

La liste des commandes

• Réglez l'objet *umenu* attaché au *bpatcher* sur «one». L'objet *bpatcher* affichera une boîte de *messages* contenant une série de messages séparés par des virgules. Cliquez sur la boîte de *messages* pour envoyer son contenu à notre objet *jit.gl.sketch*.

Un triangle vert devrait apparaître dans l'objet *jit.pwindow*.


Un triangle vert dessiné avec une séquence de messages à *jit.gl.sketch*.

> **Note technique**: l'objet *jit.gl.sketch* fonctionne en maintenant une liste de commandes d'OpenGL qui sont ensuite exécutées à chaque fois que la scène est dessinée (dans notre patch, lorsque l'objet *jit.gl.render* reçoit un **bang**). La plupart des messages que nous envoyons à *jit.gl.sketch* font partie de cette liste de commandes. Cette liste de commandes peut être stockée dans la mémoire interne de l'objet ou, si l'attribut **displaylist** de *jit.gl.sketch* est défini sur **1**, elle peut être stocké sur le GPU de notre ordinateur. Lorsque l'on travaille avec un grand nombre de commandes, l'activation de l'attribut **displaylist** permet d'accélérer le temps de rendu; cependant, les nouvelles commandes ajoutées à la liste peuvent prendre plus de temps si elles sont envoyées très rapidement, car elles doivent être chargées sur le GPU.

Notre première boîte de *message* envoie les messages suivants dans l'ordre à *jit.gl.sketch*:

reset,
glcolor 0 1 0 1,
glbegin line_loop,
glvertex -0,5 -0,5,
glvertex 0 0,5,
glvertex 0,5 -0,5,
glend

Le message **reset** de *jit.gl.sketch* indique simplement à l'objet d'effacer sa liste de commandes et de se réinitialiser. Ce qui suit est une séquence de commandes OpenGL. La commande **glcolor** indique à *jit.gl.sketch* de changer la couleur qu'il utilise pour dessiner. Comme pour tous les objets OpenGL dans Jitter, ces couleurs sont des valeurs à virgule flottante dans l'ordre RGBA (rouge, vert, bleu, alpha); ainsi, **glcolor 0 1 0 1** indique à *jit.gl.sketch* de dessiner en vert entièrement opaque (alpha = 1).

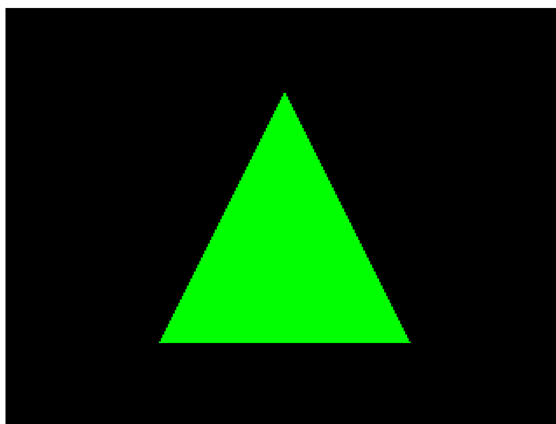
Le message **glbegin** indique à *jit.gl.sketch* d'interpréter ce qui suit comme des instructions définissant un objet. L'argument de **glbegin** spécifie la ***primitive*** de dessin à utiliser lors du rendu de la forme. La primitive que nous avons choisie, **line_loop**, prendra un ensemble de points (appelés sommets) et les connectera avec des lignes, complétant la forme en connectant le dernier sommet au premier sommet à nouveau. Ainsi, dans une forme avec quatre points A, B, C et D, notre objet contiendra des lignes de A à B, B à C, C à D et D à A.

Les messages **glvertex** contiennent les coordonnées des points de notre forme. Si seulement deux coordonnées sont données, elles sont interprétées comme des valeurs x et y. Si trois coordonnées sont données, le dernier point est interprété comme un point de référence. Si trois coordonnées sont données, la dernière valeur est interprétée comme une coordonnée z. Le message **glend** indique à *jit.gl.sketch* que nous avons fini de définir la forme. Nous pouvons alors passer à une autre forme (ou exécuter d'autres commandes) si nous le souhaitons.

La première étape de notre patch réinitialise donc l'objet, lui donne la couleur verte et lui demande de dessiner une forme définie avec trois sommets reliés entre eux. Ces messages (à l'exception du message **reset**) forment la liste de commandes pour *jit.gl.sketch* pour l'image que nous voyons dans *jit.pwindow*.

En savoir plus sur les primitives de dessin

• Sélectionnez l'objet *umenu* et attribuez-lui la valeur «deux». Cliquez sur la boîte de *message* qui apparaît dans le *bpatcher*. Notre triangle encadré devrait disparaître et un triangle en vert uni devrait prendre sa place.



Un triangle vert plein.

Un triangle vert uni.

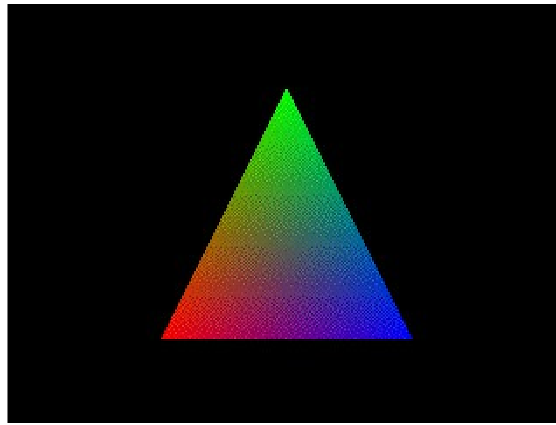
La liste des commandes dans notre boîte de *message* est très similaire à la première:

reset,
glcolor 0 1 0 1,
triangles de glbegin,
glvertex -0,5 -0,5,
glvertex 0 0,5,
glvertex 0,5 -0,5,
glend

Une fois encore, nous réinitialisons l'objet, puis nous définissons une couleur (vert) et un objet avec trois sommets à dessiner. La seule différence est la primitive de dessin spécifiée en tant qu'argument du message **glbegin**. Dans cet exemple, nous utilisons des **triangles** comme primitive de dessin. Cela indique à *jit.gl.sketch* d'interpréter les triades de sommets comme des triangles. Plutôt que de tracer le contour de la forme comme nous l'avons fait auparavant, nous avons maintenant demandé à *jit.gl.sketch* de générer un polygone. En conséquence, l'intérieur de nos sommets est rempli de la **glcolor** spécifiée (vert).

> Il existe dix (10) primitives de dessin différentes reconnues par OpenGL et, par conséquent, par des objets tels que *jit.gl.render* et *jit.gl.sketch*. Ce sont les *points*, les *lignes*, *line_strip*, *line_loop*, les *triangles*, *triangle_strip* (en abrégé tri_strip), *triangle_fan* (en abrégé *tri_fan*), les *quads*, *quad_strip* et *polygone*. Ces primitives spécifient chacune l'algorithme par lequel une série de sommets sera connectée. Le ***Tutoriel 37***: ***Géométrie sous le chapeau*** et le tableau 2-2 et la figure 2-7 du «Redbook» OpenGL donnent des descriptions et des exemples de chacune d'entre elles et décrivent leurs utilisations courantes.

• Définissez l'objet *umenu* dans le patch de Tutorial sur «three» et cliquez sur la boîte de *message* qui apparaît dans le *bpatcher*. Un triangle avec une surface arc-en-ciel apparaîtra à la place du triangle vert uni.



Une forme avec différents sommets colorés.

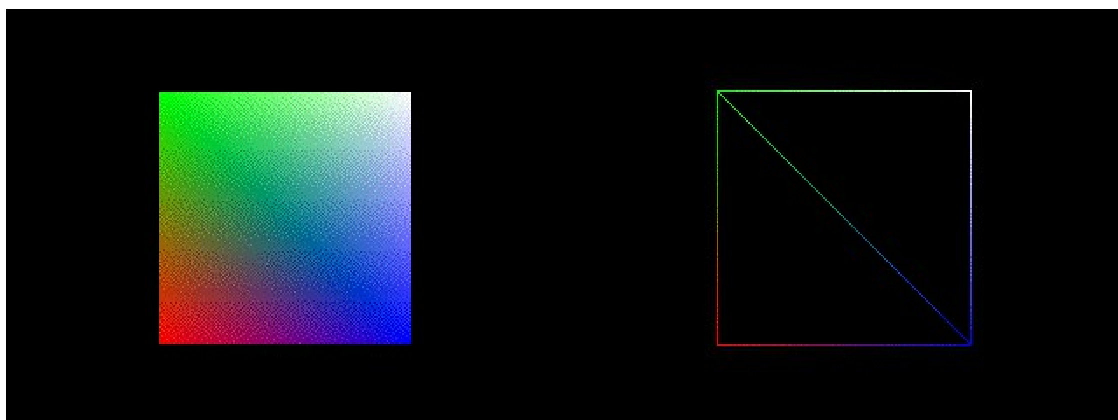
Une forme avec des sommets de couleurs différentes.

Dans cette liste de commandes, nous avons intégré des commandes **glcolor** pour chaque sommet de notre triangle:

```
reset,
triangles de glbegin,
glcolor 1 0 0 1,
glvertex -0,5 -0,5,
glcolor 0 1 0 1,
glvertex 0 0,5,
glcolor 0 0 1 1,
glvertex 0,5 -0,5,
glend
```

Par conséquent, notre objet *jit.gl.sketch* définit nos trois sommets en rouge, vert et bleu, respectivement, et remplit la surface intérieure avec un dégradé de couleurs qui s'estompe progressivement entre les trois couleurs. L'attribut **smooth_shading** d e *jit.gl.sketch* permet ce comportement lorsqu'il est défini sur **1** (comme c'est le cas avec l'objet jit.gl.sketch dans notre patch de Tutoriel).

• Donnez la valeur «four» à l'objet *umenu* dans le patch de Tutoriel. Réglez *l'umenu* dans le *bpatcher* sur "fill". Notre *jit.pwindow* contient maintenant un carré avec une surface aux couleurs de l'arc-en-ciel. Réglez *l'umenu* sur «outline». Maintenant, nous voyons seulement des lignes aux couleurs de l'arc-en-ciel qui délimitent la forme, ainsi qu'une diagonale qui la traverse.



Un carré composé de triangles connectés, remplis et non remplis.

Cette section du *bpatcher* nous permet de voir comment une primitive de dessin fonctionne pour réaliser une forme plus complexe. Nous commençons notre liste de commandes par la commande **glpolygonmode**, qui indique à *jit.gl.sketch* s'il faut remplir des polygones complets ou les laisser sous forme de contours, exposant le squelette qui définit comment les sommets sont connectés. Le premier argument de **glpolygonmode** définit si nous parlons de l'avant de la forme, de l'arrière de la forme ou des deux (**front_and_back**, comme nous l'utilisons ici). Le deuxième argument définit si ce côté de la forme sera plein (**fill**) ou souligné (**line**). L'attribut **poly_mode** des objets Jitter OpenGL accomplit la même chose.

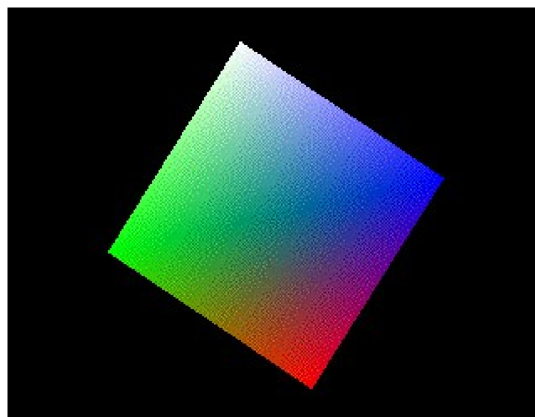
Après avoir réinitialisé et décidé de la façon dont nous voulons que notre forme soit dessinée (remplie ou esquissée), nous fournissons une description de la forme:

```
glbegin tri_strip,
glcolor 1 0 0 1,
glvertex -0,5 -0,5,
glcolor 0 1 0 1,
glvertex -0,5 0,5,
glcolor 0 0 1 1,
glvertex 0,5 -0,5,
glcolor 1 1 1 1,
glvertex 0,5 0,5,
glend
```

Comme pour le triangle dans notre exemple précédent, nous fournissons une **glcolor** pour chaque **glvertex** (dans cet exemple rouge, vert, bleu et blanc pour les quatre points du carré). La primitive de dessin **tri_strip** utilisée comme argument pour **glbegin** dit à *jit.gl.sketch* de dessiner la forme comme une série de triangles où les deux derniers sommets d'un triangle sont recyclés dans le triangle suivant. Ainsi, un polygone avec six sommets A, B, C, D, E et F serait dessiné comme quatre triangles en utilisant la primitive **tri_strip**: ABC, CBD, CDE et EDF (l'ordre garantit que les triangles sont tous dessinés avec la même orientation). Dans notre code, nous obtenons une forme avec deux triangles qui, une foist connectés et remplis, nous apparaissent comme un carré.

Rotation, translation et mise à l'échelle

• Réglez *l'umenu* du patch de Tutorial sur «five». Cliquez et faites glisser la boîte de *nombres* du *bpatcher* lentement vers le haut, de **0** à **360**. Le carré arc-en-ciel plein de l'exemple précédent réapparaîtra et tournera dans le sens inverse des aiguilles d'une montre.



Une matrice de rotation appliquée à un objet.

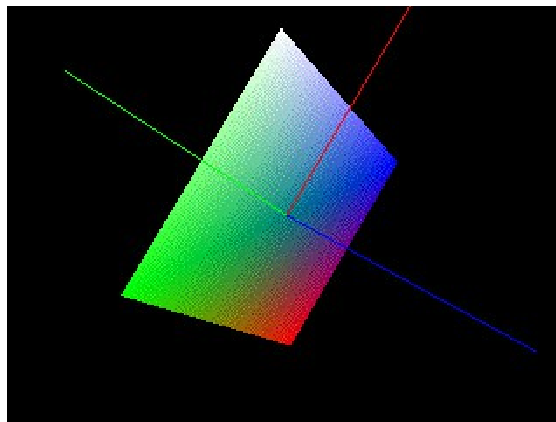
En plus de définir des formes contenant des sommets et des couleurs, nous pouvons demander à *jit.gl.sketch* de manipuler ces formes comme des objets pouvant être animés et mis à l'échelle. Notre liste de commandes met cela en place pour nous, en fournissant des commandes supplémentaires:

reset,
glmatrixmode modelview,
glpushmatrix,
glrotate 1 0 0 1 $,

Après avoir tout réinitialisé, nous envoyons le message **glmatrixmode**, qui indique à *jit.gl.sketch* que nous allons effectuer des opérations de perspective pour définir quelque chose appelé la matrice modelview. Cela nous permet d'effectuer une rotation, une translation et une mise à l'échelle sur l'objet. Le message **glpushmatrix** copie notre matrice modelview sur une pile, nous permettant d'y revenir plus tard. Cela nous permet de créer des translations imbriquées pour dessiner des objets complexes. Par exemple, nous pourrions dessiner une série de polygones, chacun avec des rotations par rapport au polygone ***précédent***; de cette façon, nous pourrions générer des séquences de formes qui ont la même orientation les unes par rapport aux autres, mais pourraient alors toutes être en rotation par rapport à un autre groupe de formes. L'utilisation d'une pile pour ce processus facilite grandement la modélisation, car nous pouvons utiliser de simples coordonnées face à face pour les sommets de nos formes, puis dire au moteur de rendu de calculer les rotations pour nous. La commande **glrotate** spécifie les ***degrés*** de rotation de notre forme suivis de trois valeurs qui spécifient l'axe de rotation comme un vecteur xyz. Ainsi, nous faisons tourner notre objet autour de l'axe z (qui fait tourner la forme le long du plan x-y de l'écran).

Après la rotation, nous continuons avec notre liste de commandes comme dans l'exemple précédent. À la fin de la spécification de la forme, nous complétons notre liste de commandes avec la commande **glpopmatrix**, qui ramène notre objet *jit.gl.sketch* à la rotation précédente. Les nouvelles commandes ajoutées à la liste de commandes après cela utiliseraient alors le vecteur de rotation original, et non celui que nous avons appliqué à notre forme.

• Réglez *l'umenu* sur «six» et manipulez les objets de la boîte de *nombre* attachés à l'objet *pak* à l'intérieur du *bpatcher*. Nous pouvons maintenant faire pivoter notre carré autour des trois axes. De plus, des lignes sont rendues pour nous fournir des vecteurs unitaires pour nous guider dans notre rotation.



Une matrice de rotation appliquée à plusieurs objets (un carré et trois vecteurs).

Notre liste de commandes établit ici une nouvelle matrice **modelview** et effectue la rotation trois fois en série:

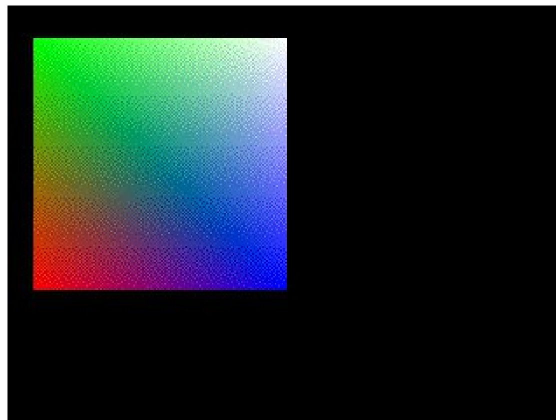
reset,
glmatrixmode modelview,
glpushmatrix,
glrotate 1 $ 1 0 0,
glrotate 2 0 1 0 $,
glrotate 3 0 0 1 $,

Cela nous permet de faire pivoter notre forme le long de l'axe x, de l'axe y et de l'axe z en séquence en fonction des valeurs des objets de boîte de *nombre* dans notre patch. Nous créons ensuite quatre objets en utilisant cette rotation avant d'envoyer une **glpopmatrix** pour revenir à notre orientation initiale. Après le rendu de notre carré, nous créons trois lignes à deux sommets de couleurs différentes orientées le long des mêmes axes que le plan:

glcolor 1 0 0 1, glbegin lines, glvertex 0 0 0, glvertex 1 0 0, glend,
glcolor 0 1 0 1, glbegin lines, glvertex 0 0 0, glvertex 0 1 0, glend,
glcolor 0 0 1 1, lignes glbegin, glvertex 0 0 0, glvertex 0 0 1, glend, \ nglpopmatrix

Le dessin des **lignes** primitives relie simplement tous les sommets de la forme (contrairement à **line_loop**, il ne relie pas le dernier sommet au premier). Remarquez que, comme ces objets (et notre carré) sont tous dans la même matrice **modelview**, ils partagent tous la même rotation.

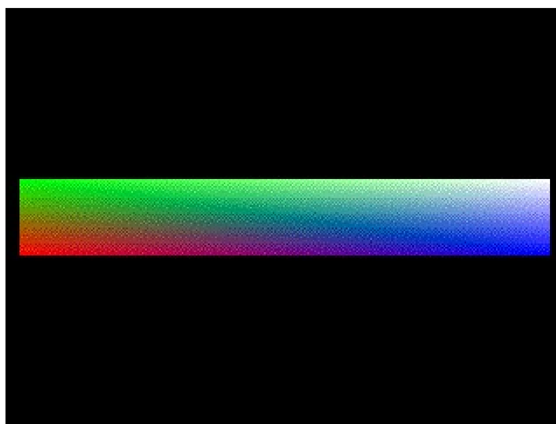
• Réglez *l'umenu* sur «seven» et manipulez les objets de la boîte de *nombre* dans le *bpatcher*. Le carré coloré peut être déplacé le long des axes x et y de l'espace.



Une matrice de translation appliquée à notre forme.

En plus de la rotation de l'image, la matrice **modelview** peut appliquer la translation d'une image. Cela nous permet de spécifier des sommets pour notre objet centré autour de (0, 0) et ensuite de déplacer l'objet où nous voulons dans l'espace. Le message **gltranslate** dans notre liste de commandes exécute cette fonction, prenant comme arguments les décalages x et y qui seront appliqués à chaque sommet de la ou les formes affectées par la transformation.

• Réglez *l'umenu* sur «eight» et manipulez les objets de la boîte de *nombre* dans le *bpatcher*. Le carré coloré peut être étiré le long des deux axes.



Notre objet transformé par mise à l'échelle.

Tout comme nous pouvons faire pivoter et translater une ou plusieurs formes, nous pouvons également contrôler leur mise à l'échelle avec le message **glscale**. Les sommets de notre forme sont multipliés par les arguments du message **glscale** (spécifiés en tant que facteurs d'échelle x, y et, en option,z).

> L'ordre dans lequel vous fournissez les commandes **glrotate**, **gltranslate** et **glscale** à *jit.gl.sketch* est important, car elles sont traitées ***séquentiellement*** et ***cumulativement*** pour créer la transformation finale des objets affectés. Par exemple, supposons que vous ayez un sommet aux coordonnées (0,5, 1). Si vous translater ce sommet par (0,2, 0,2) puis le mettez à l'échelle par (1,5, 1,5), le sommet se retrouvera à (1,6, 1,7). Si vous inversez l'ordre de ces opérations (mise à l'échelle ***puis*** translation), votre sommet se terminera à (0,95, 1,7).

Résumé

L'objet jit.gl.*sketch* nous donne accès à la puissante API OpenGL, nous permettant de définir et d'exécuter une liste de commandes OpenGL dans un patch Max. Les listes de sommets, spécifiées par les messages **glvertex** et mises entre parenthèses par les messages **glbegin** et **glend**, peuvent être utilisées pour spécifier les formes des objets. La commande **glbegin** prend comme argument une primitive de dessin qui définit l'algorithme par lequel les sommets sont connectés et remplis. Vous pouvez utiliser la commande **glpolygonmode** pour exposer la structure des contours d'une forme, et vous pouvez colorer des formes ou des sommets entiers à l'aide de messages **glcolor**. Vous pouvez effectuer une rotation, une translation et une mise à l'échelle d'une forme définie en envoyant la commande **glmatrixview** avec l'argument **modelview**. En utilisant les commandes **glrotate**, **gltranslate** et **glscale**, vous pouvez ensuite spécifier une transformation de visualisation. Les commandes **glpushmatrix** et **glpopmatrix** vous permettent de définir une ***pile*** de ces transformations afin que vous puissiez modifier la rotation, la position et la taille des formes, soit de manière indépendante, soit en groupes imbriqués dans une séquence de commandes OpenGL.