FPS 80 (First Person Shooter for the TIC-80)

Bruno Oliveira <brunotc@gmail.com>, 2017-10-08

Formula for projecting a 3D point x,y,z onto the screen assuming camera can only rotate about Y axis (yaw only)

Parameters:

asp = aspect ratio of screen
fovy = vertical field-of-view angle
nc, fc = near and far clipping planes

ex, ey, ez = eye position in world
ϕ = camera yaw angle (rotation about Y axis)

Assuming model matrix is identity, so we have to calculate the view (V) and projection (P) matrices. For the view matrix V, this is just R*T where R=RotationMatrix([0,1,0],ϕ) and T=TranslationMatrix(-ex,-ey,-ez).

From the formula for the rotation matrix, using the Y axis as the rotation axis and ϕ as the rotation angle, we have:

```
R = [  cosϕ      0     sinϕ     0     ]
    [     0      1      0       0     ]
    [  -sinϕ     0     cosϕ     0     ]
    [    0       0      0       1     ]
```

And for the translation:

```
T = [ 1 0 0 -ex ]
    [ 0 1 0 -ey ]
    [ 0 0 1 -ez ]
    [ 0 0 0  1  ]
```

Multiplying:

```
RT = [  cosϕ      0     sinϕ     -excosϕ - ezsinϕ     ]
     [     0      1      0           -ey              ]
     [  -sinϕ     0     cosϕ      exsinϕ - ezcosϕ     ]
     [    0       0      0             1              ]
```

For shorthand:

LET c: cosϕ
LET s: sinϕ
LET A: -excosϕ - ezsinϕ
LET B: exsinϕ - ezcosϕ

So, using the shorthand,

```
RT = [  c      0     s     A    ]
     [  0      1     0    -ey   ]
     [  -s     0     c     B    ]
     [  0      0     0     1    ]
```

Now for the projection matrix, let's follow the implementation of gluPerspective and glFrustum. First, gluPerspective gives us:

```
    float ymax, xmax;
    float temp, temp2, temp3, temp4;
    ymax = znear * tanf(fovyInDegrees * M_PI / 360.0);
    xmax = ymax * aspectRatio;
    glhFrustumf2(matrix, -xmax, xmax, -ymax, ymax, znear, zfar);
```

So the params become:

```
    ymax:   nc * tan(fovy/2)
    xmax:   asp * ym = asp*nc*tan(fovy/2)
```

So:

```
    left:   -xmax = -asp*nc*tan(fovy/2)
    right:   xmax =  asp*nc*tan(fovy/2)
    bottom: -ymax =     -nc*tan(fovy/2)
    top:     ymax =      nc*tan(fovy/2)
```

Plug that into glFrustum to get:

```
P = [  t1/t2   0        0              0
       0       t1/t3    0              0
       0       0        (-fc-nc)/t4    (-t1*fc)/t4
       0       0        -1             0              ]
```

Where

```
    t1 = 2 * nc
    t2 = right - left =  2*asp*nc*tan(fovy/2)
    t3 = top - bottom =      2*nc*tan(fovy/2)
    t4 = fc - nc
```

So, simplifying (and using tfy=tan(fovy/2) for shorthand):

LET tfy: tan(fovy/2)
LET C: (-fc-nc)/(fc-nc) = (nc+fc)/(nc-fc)
LET D: (-2*nc*fc)/(fc-nc) = (2*nc*fc)/(nc-fc)

```
P = [  1/(asp*tfy)   0        0       0
       0             1/tfy    0       0
       0             0        C       D
       0             0        -1      0     ]
```

Now we can compute the product PV=PRT=

```
PV=   [1/(asp*tfy)       0    0     0    ]   [ c  0  s  A ]
      [          0   1/tfy    0     0    ] * [ 0  1  0 -ey]
      [          0       0    C     D    ]   [-s  0  c  B ]
      [          0       0    -1    0    ]   [ 0  0  0  1 ]
```

```
PV=  [  c/(asp*tfy)     0       s/(asp*tfy)      A/(asp*tfy)      ]
     [            0     1/tfy       0            -ey/tfy          ]
     [          -sC     0           cC           CB + D           ]
     [            s     0           -c           -B               ]
```

So to convert coordinates x,y,z,1 we just multiply PV by [x y z 1]^T.

For the TIC-80 screen, which is 240x136, with a fovy of 60 degrees, near clip of 0.1 and far clip of 1000, this evaluates to:

```
PV = [  0.9815c     0        0.9815s      0.9815A    ]
     [        0     1.7321   0           -1.7321 ey  ]
     [        s     0        -c          -B-0.2      ]
     [        s     0        -c          -B          ]
```

Hence:

```
local px = x*c*0.9815 + z*s*0.9815 + 0.9815*a
local py = 1.7321*y - 1.7321*ey
local pz = x*s - c*z - b - 0.2
local pw = x*s - c*z - b
local ndcx=px/pw
local ndcy=py/pw
```

```
return 120+ndcx*120,68-ndcy*68,pz
```

Technique for rendering a flat level

Assumption: level is made of rectangular walls, all vertical (no slant), same height, flat floor and flat ceiling (no changes in floor/ceiling height). On any vertical line of the screen, no more than one wall will appear (no walls above other walls).

Wall:

- Permanent data:
  - x, z world position of the 2 verts (left and right). Y coords of the wall are constants (since all have the same height).
  - texture ID
- Computed at render time:
  - **xmin, xmax:** screen space xmin, xmax. Always ordered, xmin <= xmax.
  - **zmin, zmax:** min/max z coordinates
  - **left_ymin, left_ymax:** screen space Y range at xmin
  - **right_ymin, right_ymax:** screen space Y range at xmax

So, to render the walls, we just compute the list of all walls that are potentially visible, based on the player's position and maybe the direction they are looking at (in some cheap way, like dot product between the player-to-wall vector and the camera forward vector). This is the **PVS**, potentially visible set.

For each wall in the PVS:

- compute the screen space coords: xmin, xmax, left_ymin, left_ymax, right_ymin, right_ymax from the wall's 4 world vertices.
- cull if it's completely outside of the clipping planes (xmin,xmax < 0 or > SCRW, or if zmin and zmax are outside of the near/far clipping planes)
- cull if it has its back side to us (xmax < xmin).
- for x = max(0,xmin) to min(SCRW-1,xmax)
  - compute depth, interpolate from zmin,zmax
  - if hmap[x].depth > depth: write to hmap[x]:
    - hmap[x].wall = this wall
    - hmap[x].depth = depth

At the end of this, the hmap will indicate exactly what wall to render at each X position. Now we need to go and render.

- for x = 0 to SCRW-1
    - if hmap[x] doesn't have a wall, skip
    - let w = hmap[x].wall
    - using interpolation, find the screen-space Y range for this x position
        - ymax = interpolate between w.left_ymax and w.right_ymax
        - ymin = interpolate between w.left_ymin and w.right_ymin
    - write that range to hmap[x].ymin, hmap[x].ymax, for later.
    - let u = (x - w.xmin) / (w.xmax - w.xmin)
    - for y = ymin to ymax
        - set v = (y - ymin) / (ymax / ymin)
        - sample texture at u,v
        - apply fog, light?
        - render pixel at x,y