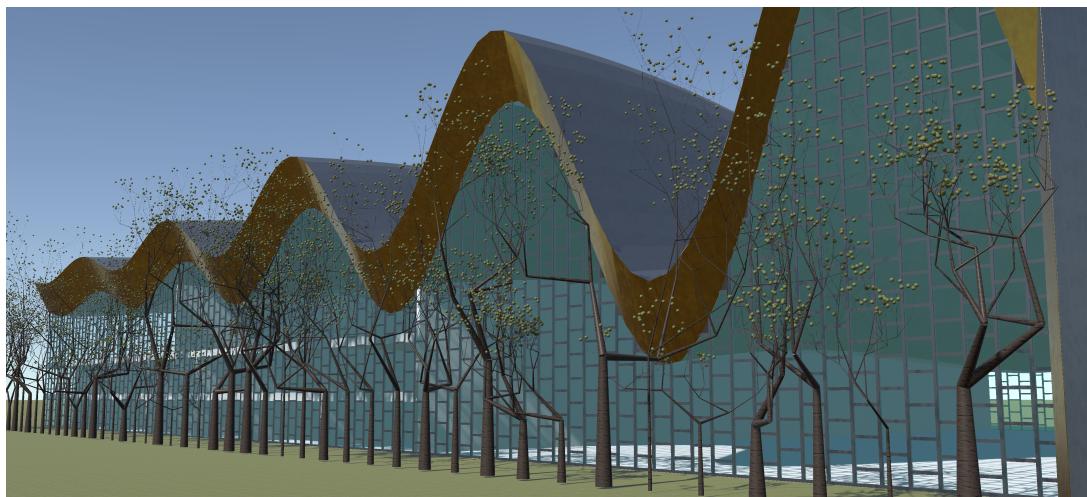


```
. using PlutoUI
```

Irina Viner-Usmanova Rhythmic Gymnastics Centre

The project presented in this notebook is an adaptation of the Irina Viner-Usmanova Rhythmic Gymnastics Centre, originally designed by CPU Pride for Moscow, Russia. Its most prominent feature is the wavy roof-structure, inspired by the movement of a rhythmic gymnastic's strip. The program elaborated here is a personal interpretation of the original design, developed in a 5 day collaborative sprint. This notebook uses the **Julia** programming language and the **Khepri** algorithmic design tool to model the building's geometry.



SET UP

```
. using WebIO
```

```
. using Khepri
```

Khepri backends size:

```
(700, 400)
```

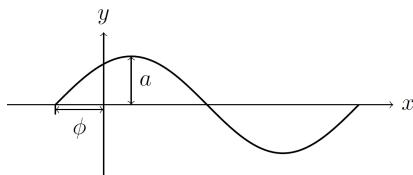
```
. render_size(700, 400)
```

Khepri render_dir command defines where the render images are saved in your PC. Use render_view to save renders to the chosen file path.

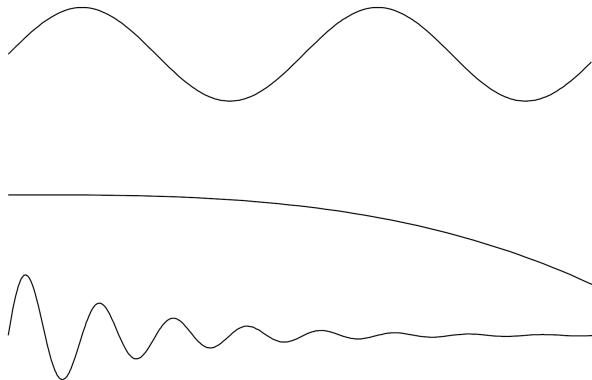
```
. # render_dir("C:\\Users\\Renata\\Documents\\GitHub\\GymnasticsCenter_Moscow\\Plots")
```

Sinusoidal Curves

Sinunoisal wave parameters



Other sinusoidal curves



sinusoidal parameters explained:

- a is the amplitude
- omega is the frequency
- fi is the phase

sinusoidal (generic function with 1 method)

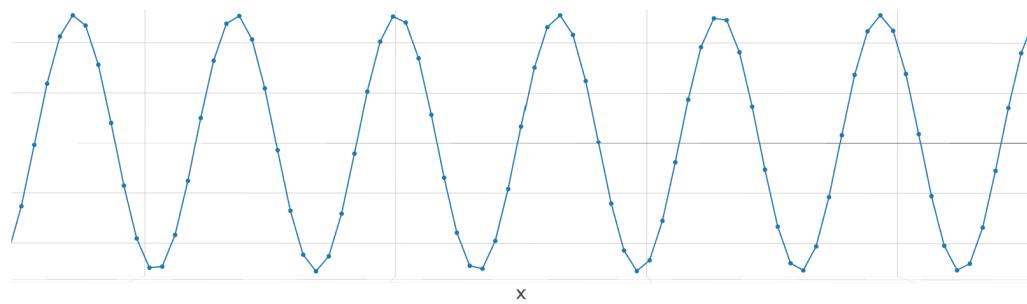
```
. sinusoidal(a, omega, fi, x) = a*sin(omega*x+fi)
```

sin_array_y (generic function with 1 method)

```
. sin_array_y(p, a, omega, fi, dist, n) = [p+vxy(i, sinusoidal(a, omega, fi, i)) for i in division(0, dist, n)]
```

```
. #=
begin
    backend(notebook)
    new_backend()
    line(sin_array_y(u0(), 5, 1, 0, 50, 100))
end
#=
```

Expected result:



damped_sin_wave's parameters explained

- a is the initial amplitude (the highest peak)
- b is the decay constant
- omega is the angular frequency

damped_sin_wave (generic function with 1 method)

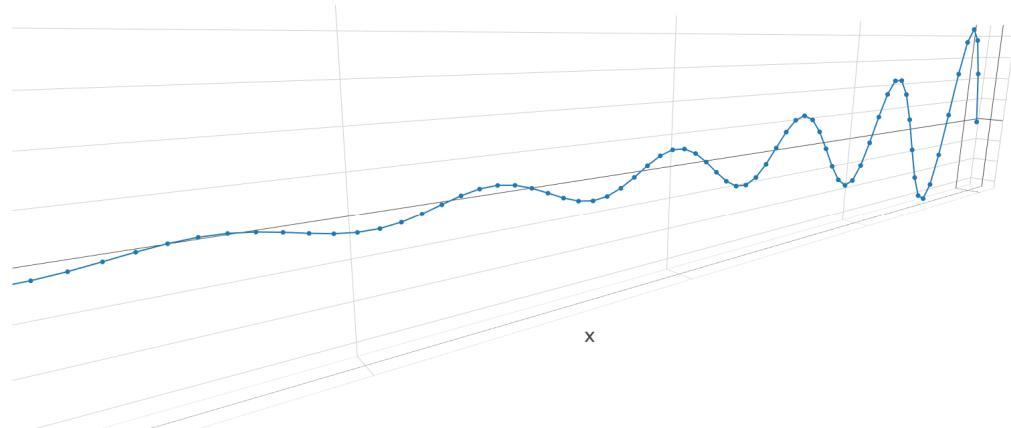
```
. damped_sin_wave(a, d, omega, x) = a*exp(-(d*x))*sin(omega*x)
```

damped_sin_array_z (generic function with 1 method)

```
. damped_sin_array_z(p, a, d, omega, dist, n) = [p+vxz(i, damped_sin_wave(a, d, omega, i)) for i in division(0, dist, n)]
```

```
. #=
. begin
.     backend(notebook)
.     new_backend()
.     line(damped_sin_array_z(u0(), 5, 0.1, 1, 50, 100))
. end
.=#
```

Expected result:



damped_sin_roof_pts (generic function with 1 method)

```

. damped_sin_roof_pts(p, h, a_x, a_y_min, a_y_max, fi, decay, om_x, om_y, dist_x, dist_y, n_x, n_y) =
.   map_division((x, y) ->
.     y <= d_i ?
.       p + vxyz(x,
.         -sin(y/d_i*(1*pi)),
.         y*h/d_i + sin(x/dist_x*pi)*sinusoidal(a_x, om_x, fi-y*pi/dist_y, x)*
.         (y*a_x/d_i)) :
.       p + vxyz(x,
.         y,
.         h + sin(x/dist_x*pi)*sinusoidal(a_x, om_x, fi-y*pi/dist_y, x) +
.         damped_sin_wave(a_y_max - (a_y_max-a_y_min)/dist_x*x, decay, om_y, y)),
.     0, dist_x, n_x,
.     0, dist_y, n_y)

```

`d_i` is the distance between the pavilion starting point and the beginning of the dumped sine curve.

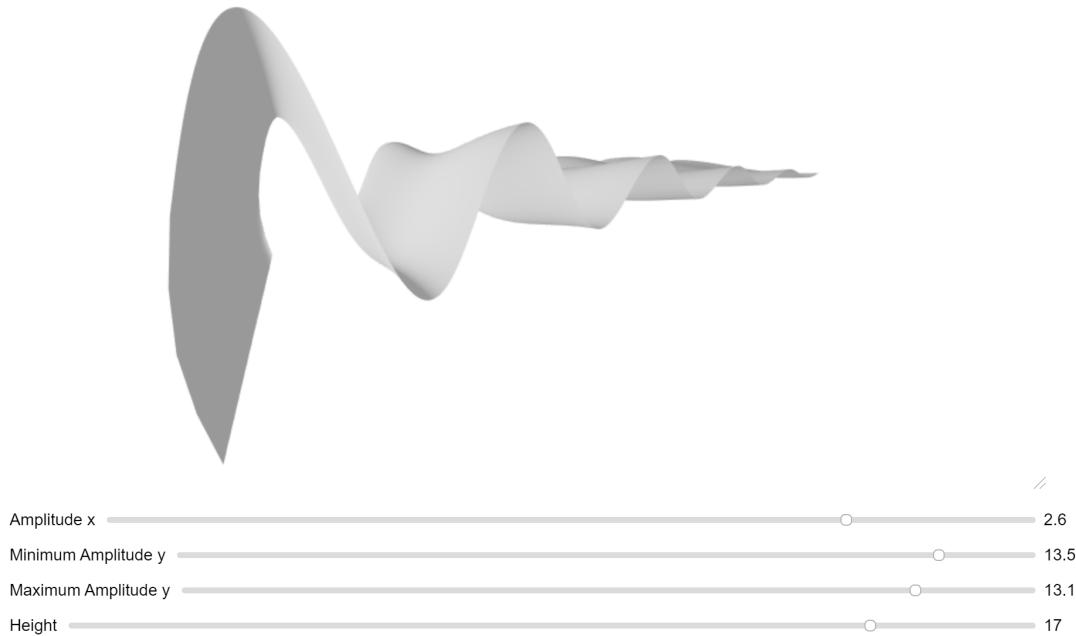
```

. #=
. begin
.   backend(meshcat)
.   new_backend()
. end
. =#

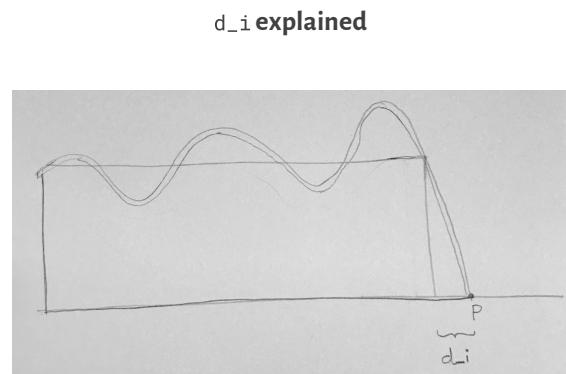
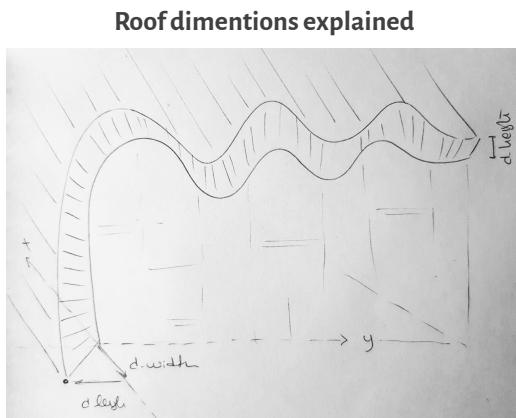
```

```
. # surface_grid(damped_sin_roof_pts(u0(), 20, 3, 10, 15, pi, 0.03, pi/50, pi/10, 60, 100, 120, 800))
```

Expected result:



Pavilion Dimensions



Double sinusoid parameters:

- amp_x = amplitude of the sinusoid along the x axis
- ampymin_top = minimum amplitude of the damped sinusoid along the y axis for the top layer of the roof
- ampmmax_top = maximum amplitude of the damped sinusoid along the y axis for the top layer of the roof
- ampmmax_bottom = maximum amplitude of the damped sinusoid along the y axis for the bottom layer of the roof
- ampymin_bottom = minimum amplitude of the damped sinusoid along the y axis for the bottom layer of the roof
- fi = sinusoid's phase along the x axis
- decay = damped sinusoid's decay along the y axis
- om_x = frequency of the sinusoid in x
- om_y = frequency of the damped sinusoid in y

General pavilion parameters:

- pav_width = pavilion's width (along x axis)
- pav_length = pavilion's length (along y axis)
- pav_height = pavilion's height (z axis)
- d_width = distance between roof layers in the x axis (top layer is wider)
- d_length = distance between roof layers in the y axis (top layer is longer)
- d_height = distance between roof layers in the z axis (height of the roof truss)
- d_i = distance before the damped sinusoid starts shaping the roof

Construction elements parameters:

- glasspanel/height = façade glass panel's height (width is defined by the number of points in the damped_sinusoid array)
- npanelsy = number of row subdivisions for triangular roof panels and glass wall vertical lines in length
- npanelsx = number of row subdivisions for triangular roof panels and glass wall vertical lines in width
- nglassverts = number of vertical panels per glass line
- pinwheelreclevel = number of times the pattern gets recursively subdivided
- pavslabthickness = floor slab thickness
- pavstructthickness = inner structural wall's thickness
- n_floors = number of floors inside the pavilion
- nwallin_width = number of transverse walls on each floor

- nwallin_length = number of longitudinal walls on each floor

6

```

. begin
.     amp_x = 2.5
.     amp_y_min_top = 4
.     amp_y_max_top = 11
.     amp_y_max_bottom = 10
.     amp_y_min_bottom = 3
.     fi = pi
.     decay = 0.03
.
.     pav_width = 60
.     pav_length = 100
.     pav_height = 14
.     d_width = 1.5
.     d_length = 1
.     d_height = 3
.     d_i = d_length
.
.     om_x = pi/pav_width
.     om_y = 10*pi/pav_length
.     om_y_bottom = 10*pi/(pav_length - 2*d_length)
.
.     glass_panel_height = 1
.     n_panels_y = 100
.     n_panels_x = 50
.     n_glass_verts = 10
.     pinwheel_rec_level = 3
.     pav_slab_thickness = 0.5
.     pav_struct_thickness = 0.3
.     n_floors = 3
.     n_wall_in_width = 4
.     n_wall_in_length = 6
. end

```

BIM families

BIM families for truss elements:

```

. free_node_fam = truss_node_family_element(default_truss_node_family(),
support=Khepri.truss_node_support(), radius=0.1);

.
. sup_node_fam = truss_node_family_element(default_truss_node_family(),
support=Khepri.truss_node_support(ux=true, uy=true, uz=true), radius=0.1);

```

BIM families for roof panels:

```
. roof_panel_fam = panel_family_element(default_panel_family());
```

```
. yellow_panel_fam = panel_family_element(default_panel_family());
```

Frame BIM family:

```
frame_width = 0.1
```

```
. frame_width=0.1
```

```
. frame_fam = column_family_element(default_column_family(), profile=rectangular_profile(frame_width, frame_width));
```

BIM families for walls and slabs:

```
. pav_slab_fam = slab_family_element(default_slab_family(), thickness=pav_slab_thickness);
```

```
. pav_wall_struct_fam = wall_family_element(default_wall_family(), thickness=pav_struct_thickness);
```

Ground family:

```
. ground_fam = slab_family_element(default_slab_family());
```

BIM family materials for Unity backend:

```
Dict()
```

```
. begin
.     set_backend_family(default_truss_bar_family(), unity,
.     unity_material_family("Default/Materials/Aluminum"))
.     set_backend_family(free_node_fam, unity, unity_material_family("Default/Materials/Aluminum"))
.     set_backend_family(sup_node_fam, unity, unity_material_family("materials/metal/Copper"))
.
.     set_backend_family(ground_fam, unity, unity_material_family("Default/Materials/Grass")) # -- grass
.     ground
.     # set_backend_family(ground_fam, unity, unity_material_family("Default/Materials/White")) # -- white
.     ground
.     # set_backend_family(ground_fam, unity, unity_material_family("Default/Materials/WhiteUnlit")) # -- white
.     ground no shadows
.
.     set_backend_family(default_panel_family(), unity,
.     unity_material_family("Default/Materials/GlassBlue"))
.     set_backend_family(frame_fam, unity, unity_material_family("Default/Materials/Steel"))
.     set_backend_family(roof_panel_fam, unity, unity_material_family("Default/Materials/Aluminum"))
.     set_backend_family(yellow_panel_fam, unity,
.     unity_material_family("materials/metal/YellowCopper"))
```

```
.      set_backend_family(pav_slab_fam, unity, unity_material_family("Default/Materials/Plaster"))
. end
```

Roof Truss

Basic truss elements

The following functions create truss elements:

- free_node and fixed_node create truss nodes
- bar creates truss bars
- panel creates truss panels

Truss elements:

```
fixed_node (generic function with 1 method)
. fixed_node(p) = truss_node(p, family=sup_node_fam)
```

```
free_node (generic function with 1 method)
. free_node(p) = truss_node(p, family=free_node_fam)
```

```
bar (generic function with 1 method)
. bar(p, q) = truss_bar(p, q)
```

Spatial truss

truss_ptss receives two sets of points, the bottom and top points defining the shape of the truss to create. It returns the points defining the truss structure.

```
truss_ptss (generic function with 1 method)
. truss_ptss(bottom_ptss,top_ptss) =
  length(top_ptss) == length(bottom_ptss) ?
  vcat([[pts,qts] for (pts,qts) in zip(bottom_ptss,top_ptss)]...) :
  vcat([[pts,qts] for (pts,qts) in zip(top_ptss[1:end-1],bottom_ptss)]..., [top_ptss[end]])
```

truss receives the type of truss to create, as well as both bottom and top points defining its shape. It creates a 3D spatial truss.

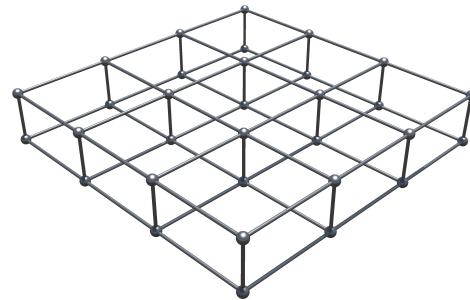
truss (generic function with 1 method)

```
. truss(truss_type; bottom_ptss=planar(p=x(0)), top_ptss=planar(p=z(1))) =
.     let ptss = truss_ptss(bottom_ptss, top_ptss)
.         truss_type(ptss)
. end
```

Vierendeel modular block front



Vierendeel modular block side



vierendeel creates a vierendeel truss structure.

vierendeel (generic function with 2 methods)

```
. vierendeel(ptss, first=true) =
.     let ais = ptss[1],
.         bis = ptss[2],
.         cis = ptss[3],
.         dis = ptss[4]
.         (first ? fixed_node : free_node).(ais)
.         free_node.(bis)
.         bar.(ais, bis)
.         bar.(ais, cis)
.         bar.(bis, dis)
.         bar.(ais[2:end], ais[1:end-1])
.         bar.(bis[2:end], bis[1:end-1])
.         if ptss[5:end] == []
.             fixed_node.(cis)
.             free_node.(dis)
.             bar.(cis[2:end], cis[1:end-1])
.             bar.(dis[2:end], dis[1:end-1])
.             bar.(dis, cis)
.         else
.             vierendeel(ptss[3:end], false)
.         end
.     end
```

```
. #
. begin
.     backend(meshcat)
.     new_backend()
```

```

. end
. =#

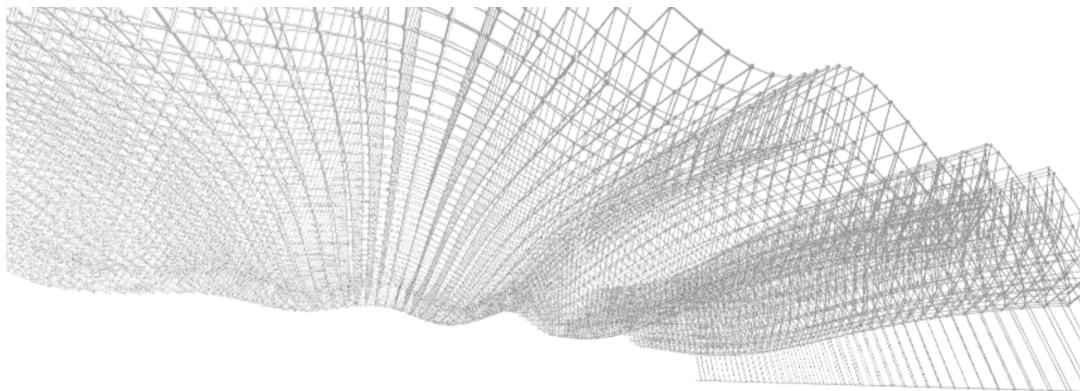
```

```

. #=
. begin
.     ampx = 3
.     ampy_top_min = 10
.     ampy_top_max = 15
.     ampy_bottom_min = 10
.     ampy_bottom_max = 12
.     h = 20
.     truss(vierendeel,
.         top_ptss=damped_sin_roof_pts(u0(), pav_height,
.                                         ampx, ampy_top_min, ampy_top_max,
.                                         fi, decay, om_x, om_y,
.                                         pav_width, pav_length-d_length, 50, 80),
.         bottom_ptss=damped_sin_roof_pts(xy(d_width,d_length), pav_height-d_height,
.                                         ampx, ampy_bottom_min, ampy_bottom_max,
.                                         fi, decay, om_x, om_y_bottom,
.                                         pav_width - d_width*2, pav_length-2*d_length, 50, 80))
.     end
. =#

```

Expected result:

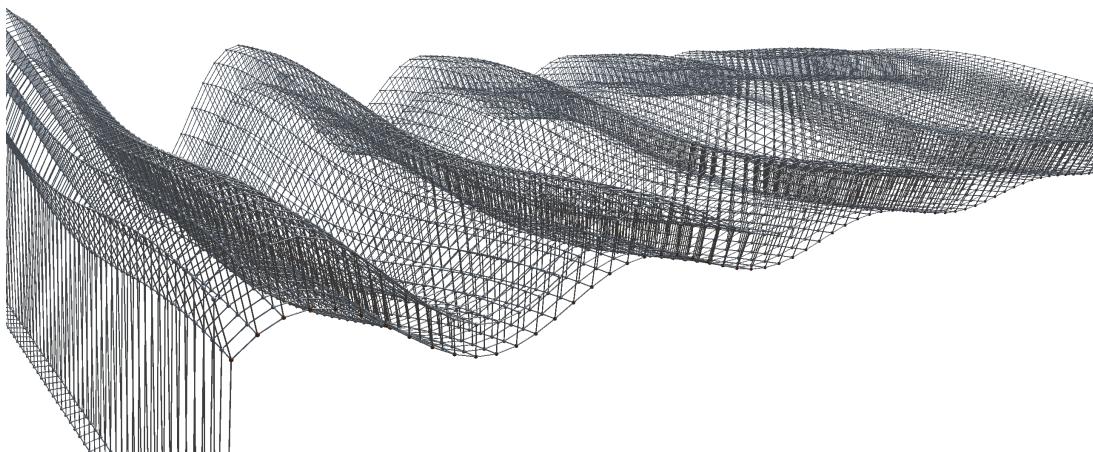


```

. #=
. begin
.     backend(unity)
.     delete_all_shapes()
.     ground()
.     truss(vierendeel,
.         top_ptss=damped_sin_roof_pts(u0(), pav_height,
.                                         amp_x, amp_y_min_top, amp_y_max_top,
.                                         fi, decay, om_x, om_y,
.                                         pav_width, pav_length-d_length, 5, 8),
.         bottom_ptss=damped_sin_roof_pts(xy(d_width,d_length), pav_height-d_height,
.                                         amp_x, amp_y_min_bottom, amp_y_max_bottom,
.                                         fi, decay, om_x, om_y_bottom,
.                                         pav_width - d_width*2, pav_length-2*d_length, 5, 8))
.     # render_view("truss_roof_unity")
.     end
. =#

```

Expected result:



Roof Surface

Auxiliar functions

`transpose_array` receives an array of arrays and transposes it.

```
transpose_array (generic function with 1 method)
· transpose_array(arrays) =
·   [[row[i] for row in arrays]
·     for i in 1:length(arrays[1])]
```

`pts_distances` receives an array of points and returns an array with the distances between ordered points.

```
pts_distances (generic function with 2 methods)
· pts_distances(pts, last_pt=true) =
·   let n = last_pt ? 0 : 1
·     [distance(p,q)
·      for (p,q) in zip(pts,[pts[2:end]...,pts[1]])][1:end-n]
·    end
```

```
shape_grid_polygon_vertices (generic function with 1 method)
· shape_grid_polygon_vertices(pts) =
·   () -> pts
```

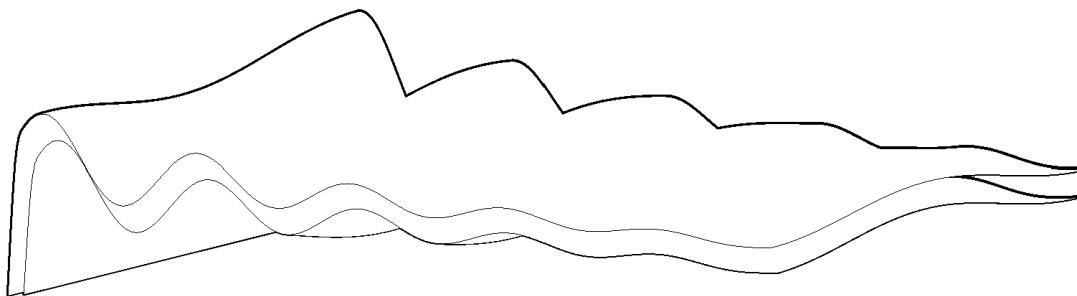
`itera_2triangs` receives an array of arrays of points. It returns the same points rearranged to create a triangular grid of points.

```
itera_2triangs (generic function with 1 method)
. itera_2triangs(ptss) =
.   vcat([vcat([[p0,p1,p3],[p1,p2,p3]]
.         for (p0,p1,p2,p3)
.           in zip(pts0[1:end-1], pts1[1:end-1], pts1[2:end], pts0[2:end])]...)]
.   for (pts0, pts1) in zip(ptss[1:end-1], ptss[2:end])]...)
```

Roof option 1: simple surfaces

```
. #=
. begin
.   backend(autocad)
.   delete_all_shapes()
.   top_roof_test = damped_sin_roof_pts(x(0), 15,
.                                         3, 6, 9,
.                                         fi, decay, om_x, om_y,
.                                         pav_width, pav_length-d_length, 50, 100)
.   bottom_roof_test = damped_sin_roof_pts(xy(d_width,d_length*1), 15-d_height,
.                                         3, 6, 8,
.                                         fi, decay, om_x, om_y_bottom,
.                                         pav_width - d_width*2, pav_length-2*d_length, 50, 100)
.   surface_grid(top_roof_test)
.   surface_grid(bottom_roof_test)
. end
.=#
```

Expected result:

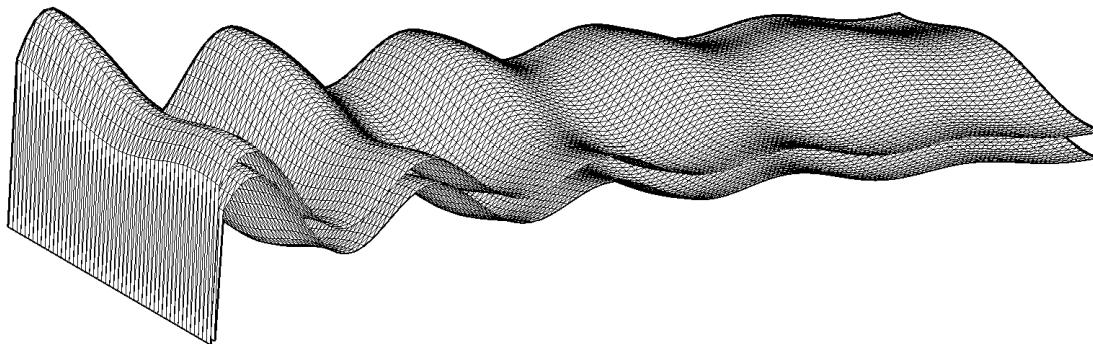


Roof option 2: triangular roof panels

CAD version: using surface_polygon subdivision

```
. #=
. begin
.   backend(autocad)
.   new_backend()
.   map(ps->surface_polygon(ps), itera_2triangs(top_roof_test))
.   map(ps->surface_polygon(ps), itera_2triangs(bottom_roof_test))
. end
.=#
```

Expected result:



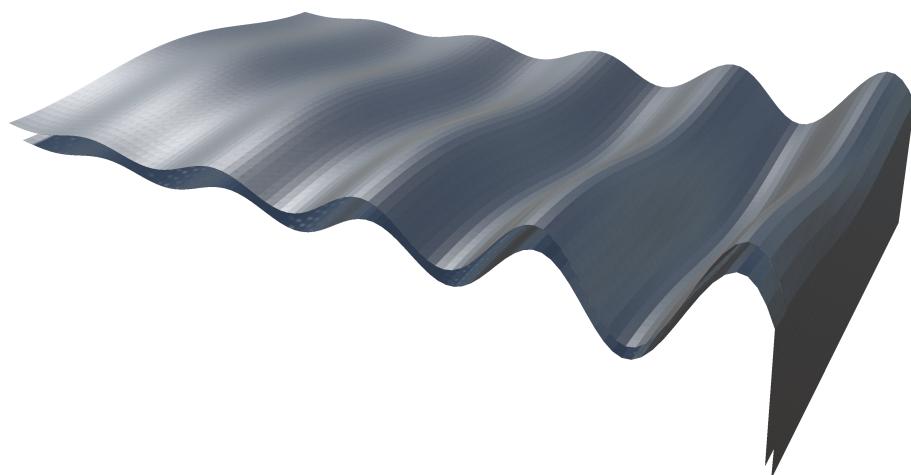
BIM version: using panel subdivision

```

. #=
. begin
.   backend(unity)
.   delete_all_shapes()
.   ground()
.   map(ps->panel(ps, family=roof_panel_fam), itera_2triangs(top_roof_test))
.   map(ps->panel(ps, family=roof_panel_fam), itera_2triangs(bottom_roof_test))
. end
. =#
. # render_view("roof_panels_unity")

```

Expected result:



Roof lateral edges option 1: simple triangular panels

```

. #=
. begin
.   backend(autocad)

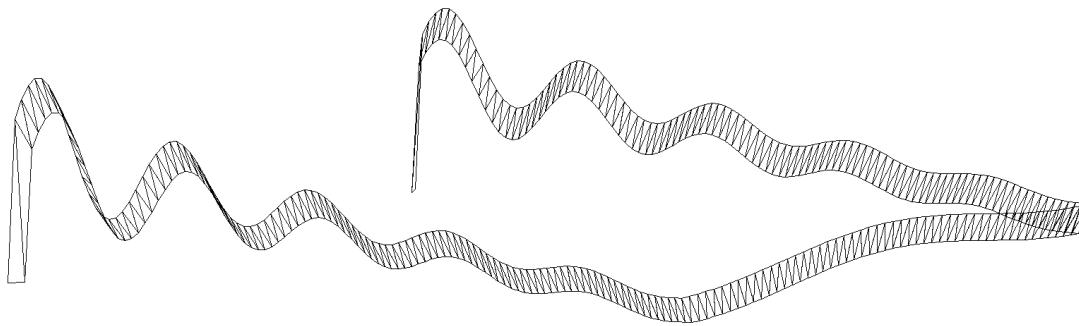
```

```

. delete_all_shapes()
. map(ps->surface_polygon(ps), itera_2triangs([top_roof_test[1],bottom_roof_test[1]]))
. map(ps->surface_polygon(ps), itera_2triangs([top_roof_test[end],bottom_roof_test[end]]))
. map(ps->surface_polygon(ps), itera_2triangs([transpose_array(top_roof_test)[end],
    transpose_array(bottom_roof_test)[end]]))
. end
. =#

```

Expected result:



Roof lateral edges option 2: Pinwheel tiling panels

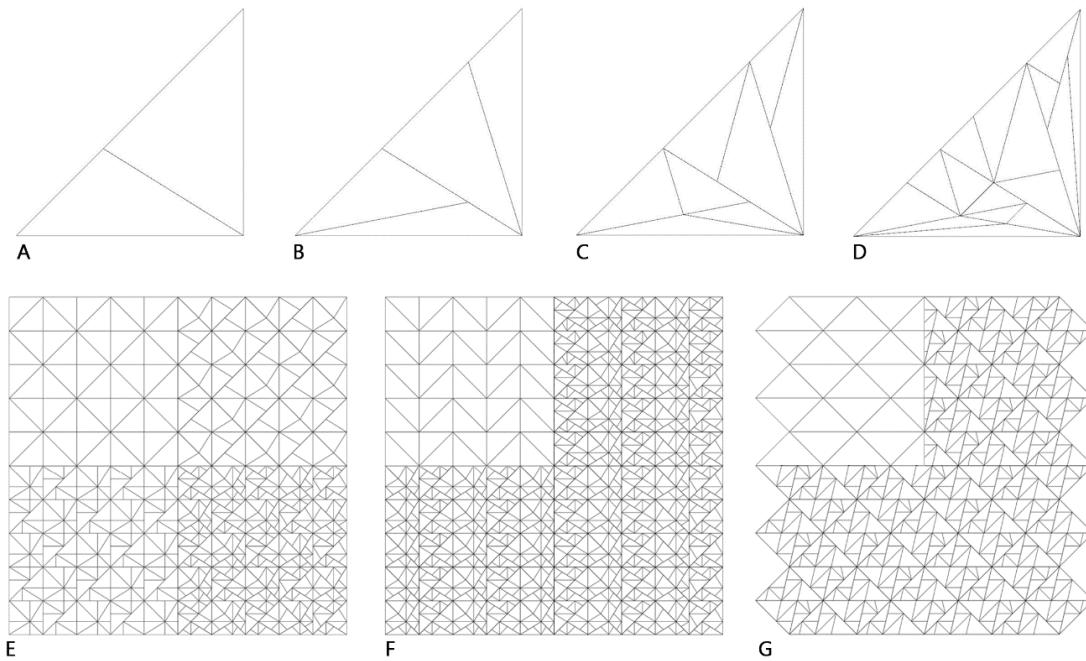
pattern creates different patterns on parametric surfaces

```

pattern (generic function with 1 method)
. pattern(fshape, pts...; args...) =
.   fshape(pts...)(; (k=>v(pts) for (k,v) in args)...)
```

pinwheel_rule defines the basic subdivision rule of the pinwheel tiling.

Pinwheel rule Tiling



pinwheel_rule (generic function with 2 methods)

```

. pinwheel_rule(pts, ratio1=0.383) =
.   let dists = pts_distances(pts)
.     max_dist = maximum(dists)
.     min_dist = minimum(dists)
.     ratio2 = 1 - ratio1
.     if max_dist == dists[1]
.       let pm = intermediate_loc(pts[1], pts[2], min_dist==dists[2] ? ratio1 : ratio2)
.         [[pts[1],pm,pts[3]],
.          [pts[2],pm,pts[3]]]
.       end
.     elseif max_dist == dists[2]
.       let pm = intermediate_loc(pts[2], pts[3], min_dist==dists[1] ? ratio1 : ratio2)
.         [[pts[1],pm,pts[2]],
.          [pts[1],pm,pts[3]]]
.       end
.     else
.       let pm = intermediate_loc(pts[1], pts[3], min_dist==dists[1] ? ratio1 : ratio2)
.         [[pts[1],pm,pts[2]],
.          [pts[2],pm,pts[3]]]
.       end
.     end
.   end
. end

```

pinwheel_recursive_rule repeats the basic subdivision rule pinwheel_rule several times (value set by the parameter level).

pinwheel_recursive_rule (generic function with 3 methods)

```

. pinwheel_recursive_rule(pts, ratio1=0.383, level=1) =
.   let qts = pinwheel_rule(pts, ratio1)
.   level == 1 ?

```

```

.     qts :
.     [qts...,
.     pinwheel_recursive_rule(qts[1], ratio1, level-1)...,
.     pinwheel_recursive_rule(qts[2], ratio1, level-1)...]
. end

```

pinwheel_tiling receives a surface (described by its points) and creates the pinwheel tiling on it.

`pinwheel_tiling` (generic function with 1 method)

```

. pinwheel_tiling(pts; ratio1=0.383, level=1) =
.     map(pts -> panel(pts, family=yellow_panel_fam), pinwheel_recursive_rule(pts, ratio1, level))

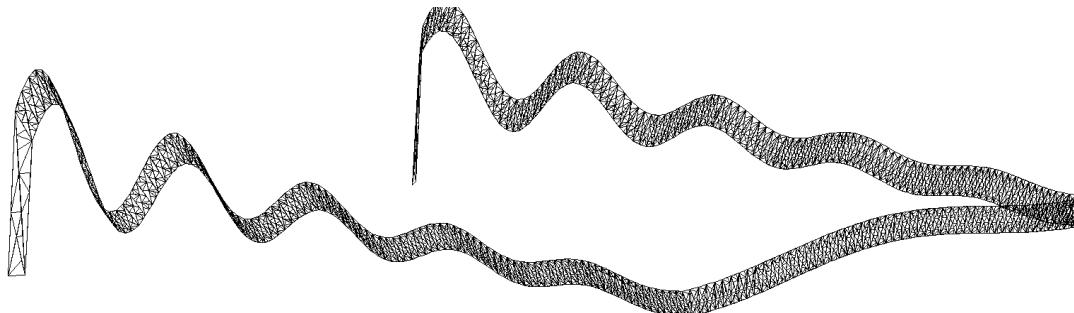
```

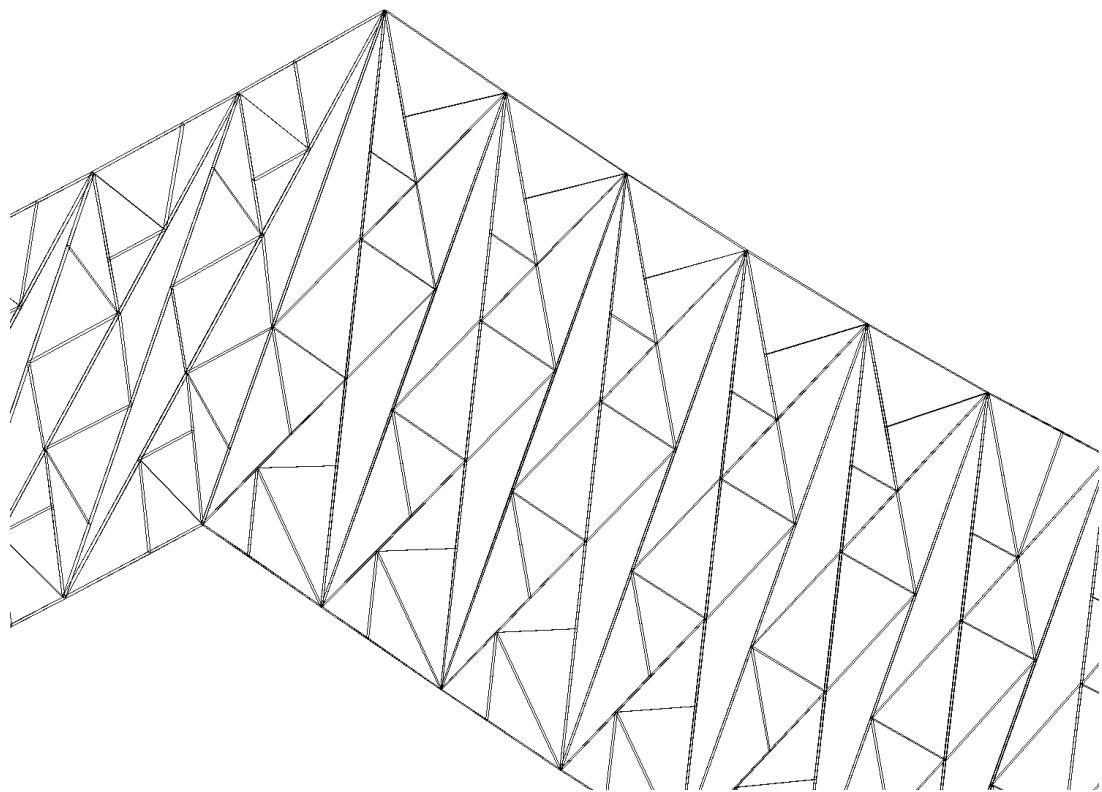
```

. #=
. begin
.     backend(autocad)
.     delete_all_shapes()
.     pinwheel_tiling.(pattern.(shape_grid_polygon_vertices,
.                             itera_2triangs([top_roof_test[1],bottom_roof_test[1]])),
.                         level=3)
.     pinwheel_tiling.(pattern.(shape_grid_polygon_vertices,
.                             itera_2triangs([top_roof_test[end],bottom_roof_test[end]])),
.                         level=3)
.     pinwheel_tiling.(pattern.(shape_grid_polygon_vertices,
.                             itera_2triangs([transpose_array(top_roof_test)
. [end], transpose_array(bottom_roof_test)[end]])),
.                         level=3)
. end
. =#

```

Expected result:





```

. #=
. begin
.     backend(unity)
.     delete_all_shapes()
.     ground()
.     pinwheel_tiling.(pattern.(shape_grid_polygon_vertices,
.                                 itera_2triangs([top_roof_test[1],bottom_roof_test[1]])),
.                           level=3)
.     pinwheel_tiling.(pattern.(shape_grid_polygon_vertices,
.                                 itera_2triangs([top_roof_test[end],bottom_roof_test[end]])),
.                           level=3)
.     pinwheel_tiling.(pattern.(shape_grid_polygon_vertices,
.                                 itera_2triangs([transpose_array(top_roof_test)
. [end], transpose_array(bottom_roof_test)[end]])),
.                           level=3)
. end
. =#

```

```
. # render_view("side_roof_panels_unity1")
```

Expected result:



Complete Roof structure

Abstracting pinwheel_tiling function for the roof sides:

```
pinwheel_broad (generic function with 1 method)
· pinwheel_broad(pts1, pts2, rec_level) =
  · pinwheel_tiling.(pattern.(shape_grid_polygon_vertices, itera_2triangs([pts1,pts2])),
    level=rec_level)
```

Roof point matrices. t_diff defines the distance between the roof surface and the truss structure within. It's currently set to 20 cm:

```
top_roof_ptss (generic function with 1 method)
· top_roof_ptss(p, n_x, n_y, truss)=
  · let dist_y = pav_length-d_length
  ·   t_diff = truss ? 0.2 : 0
  ·   damped_sin_roof_pts(p, pav_height-t_diff,
    ·                         amp_x, amp_y_min_top, amp_y_max_top,
    ·                         fi, decay, om_x, om_y,
    ·                         pav_width, dist_y, n_x, n_y)
```

```
. end
```

```
bottom_roof_pts (generic function with 1 method)
```

```
. bottom_roof_pts(p, n_x, n_y, truss)=
.   let dist_y = pav_length-d_length
.     t_diff = truss ? 0.2 : 0
.     damped_sin_roof_pts(p+vxy(d_width,d_length*1), pav_height-d_height+t_diff,
.                           amp_x, amp_y_min_top, amp_y_max_top,
.                           fi, decay, om_x, om_y_bottom,
.                           pav_width - d_width*2, pav_length-2*d_length, n_x, n_y)
. end
```

Complete roof surface with top, bottom, and side panels:

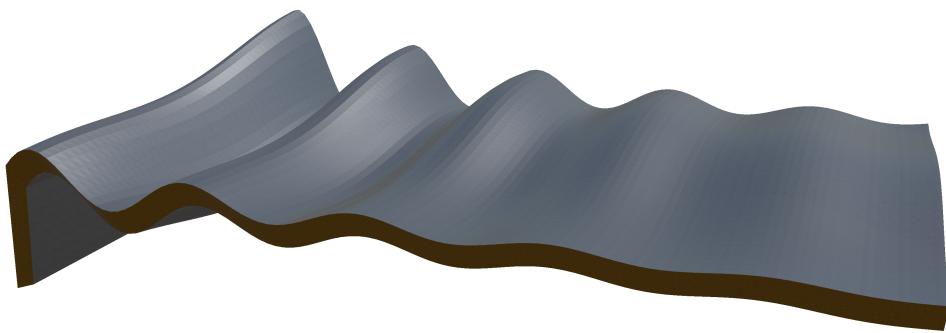
```
roof_surf (generic function with 1 method)
```

```
. roof_surf(p, n_x, n_y)=
.   let lev = pinwheel_rec_level
.     top_roof = top_roof_ptss(p, n_x, n_y, false)
.     bottom_roof = bottom_roof_pts(p, n_x, n_y, false)
.     # top and bottom aluminum roof tiles
.     map(ps->panel(ps, family=roof_panel_fam), itera_2triangs(top_roof))
.     map(ps->panel(ps, family=roof_panel_fam), itera_2triangs(bottom_roof))
.     # side yellow copper roof tiles
.     pinwheel_broad(top_roof[1], bottom_roof[1], lev)
.     pinwheel_broad(top_roof[end], bottom_roof[end], lev)
.     pinwheel_broad(transpose_array(top_roof)[end], transpose_array(bottom_roof)[end], lev)
. end
```

```
. #=
. begin
.   backend(unity)
.   delete_all_shapes()
.   ground()
.   roof_surf(u0(), 50, 100)
. end
. =#
```

```
. # render_view("complete_roof_panels_unity")
```

Expected result:



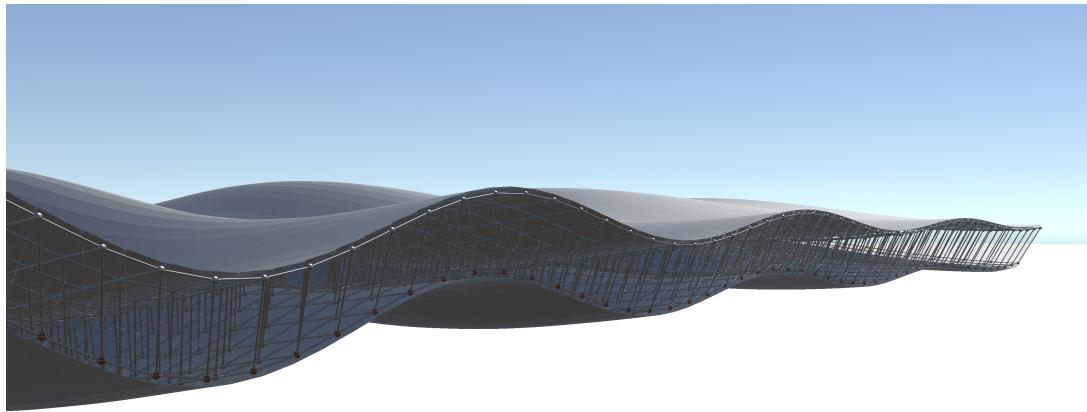
Truss structure function using the new point matrices for the pavilion:

```
roof_truss (generic function with 1 method)
· roof_truss(p, n_x, n_y)=
·     let top_roof = top_roof_ptss(p, n_x, n_y, true)
·         bottom_roof = bottom_roof_pts(p, n_x, n_y, true)
·             truss(vierendeel, top_ptss=top_roof, bottom_ptss=bottom_roof)
·     end

· #=
· begin
·     backend(unity)
·     delete_all_shapes()
·     ground()
·     set_backend_family(yellow_panel_fam, unity, unity_material_family("Default/Materials/Glass"))
·     roof_surf(u0(), 50, 100)
·     roof_truss(u0(), 30, 70)
· end
· =#
```

```
· # render_view("complete_roof_unity")
```

Expected result:



Glass Façade

splines4surf maps splines over a matrix of points, to visualise surfaces in the notebook backend faster.

splines4surf (generic function with 1 method)

```
. function splines4surf(mtx)
.     [spline(pts) for pts in mtx]
.     [spline(pts) for pts in transpose_array(mtx)]
. end
```

Wall points

damped_sin_glass_wall creates the pavilion's side glass wall matrices

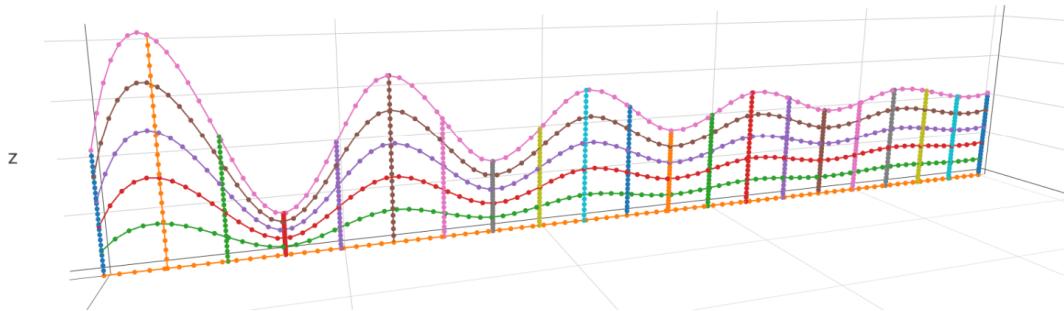
damped_sin_glass_wall (generic function with 1 method)

```
. damped_sin_glass_wall(p, a_y, fi, decay, om_y, dist_y, dist_z, n_y, n_z) =
. map_division((y, z) ->
.             y <= d_i ?
.             p + vyz(-sin(y/d_i*(1*pi)), z) :
.             p + vyz(y,
.                     z + (z/dist_z)*damped_sin_wave(a_y, decay, om_y, y)),
.             0, dist_y, n_y,
.             0, dist_z, n_z)
```

```
. #=
. begin
.     backend(notebook)
.     new_backend()
.     west_glass_wall_test = damped_sin_glass_wall(x(d_width),
.                                         amp_y_max_top,
.                                         fi, decay, om_y,
.                                         pav_length-2*d_length, pav_height-d_height, 20, 5)
.     splines4surf(west_glass_wall_test)
```

- *end*
- *=#*

Expected result:

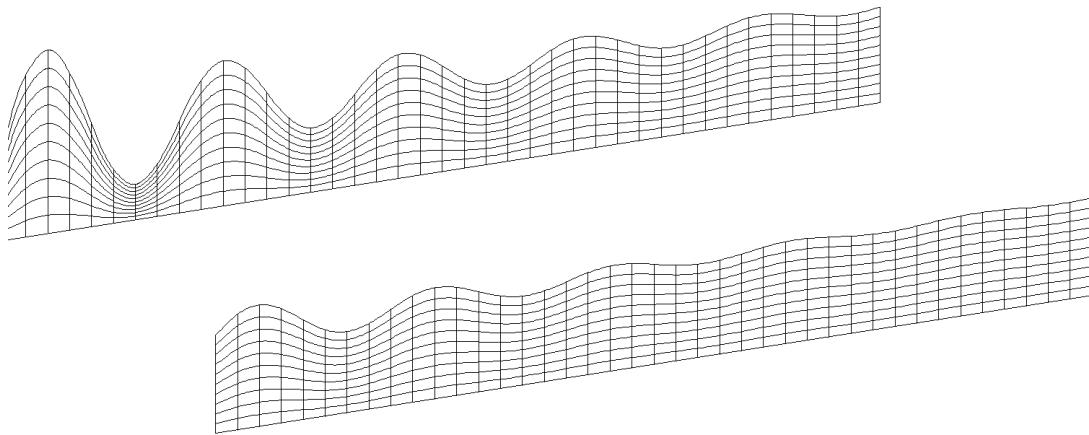


```

· #=
· begin
·     backend(autocad)
·     delete_all_shapes()
·     west_glass_wall_t1 = damped_sin_glass_wall(xy(d_width, d_length),
·                                                 amp_y_max_top,
·                                                 fi, decay, om_y_bottom,
·                                                 pav_length-2*d_length, pav_height-d_height, 40, 10)
·     east_glass_wall_t1 = damped_sin_glass_wall(xy(pav_width-d_width, d_length),
·                                                 amp_y_min_bottom,
·                                                 fi, decay, om_y_bottom,
·                                                 pav_length-2*d_length, pav_height-d_height, 40, 10)
·     splines4surf(west_glass_wall_t1)
·     splines4surf(east_glass_wall_t1)
·     #      surface_grid(bottom_roof_pts(u0()), 40, 100, false))
· end
· =#

```

Expected result:



Vertical BIM panels

Given a list of points (closed polygon vertices), this function creates a polygonal glass panel surrounded by a thin metal framing all around:

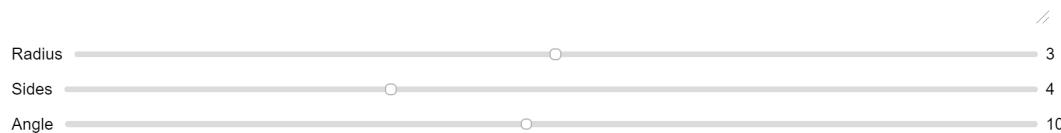
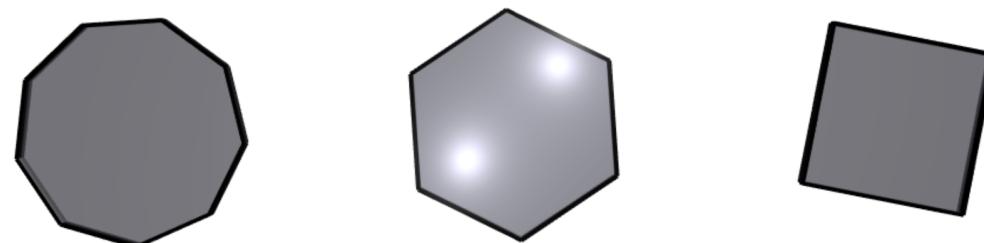
```
framed_panel (generic function with 1 method)
```

```
. framed_panel(pts)=
.   begin
.     panel(pts, family=default_panel_family())
.     for (p,q) in zip(pts, [pts[2:end]...,pts[1]])
.       free_column(p,q, family=frame_fam)
.     end
.   end
```

```
. #=
. begin
.   backend(meshcat)
.   new_backend()
. end
.=#
```

```
. #=
. begin
.   sides = 3
.   radius = 5
.   angle = pi/4
.   delete_all_shapes()
.   framed_panel(regular_polygon_vertices(sides, x(-10), radius, angle))
.   framed_panel(regular_polygon_vertices(sides+2, x(0), radius, angle))
.   framed_panel(regular_polygon_vertices(sides+5, x(10), radius, angle))
. end
.=#
```

Expected result:



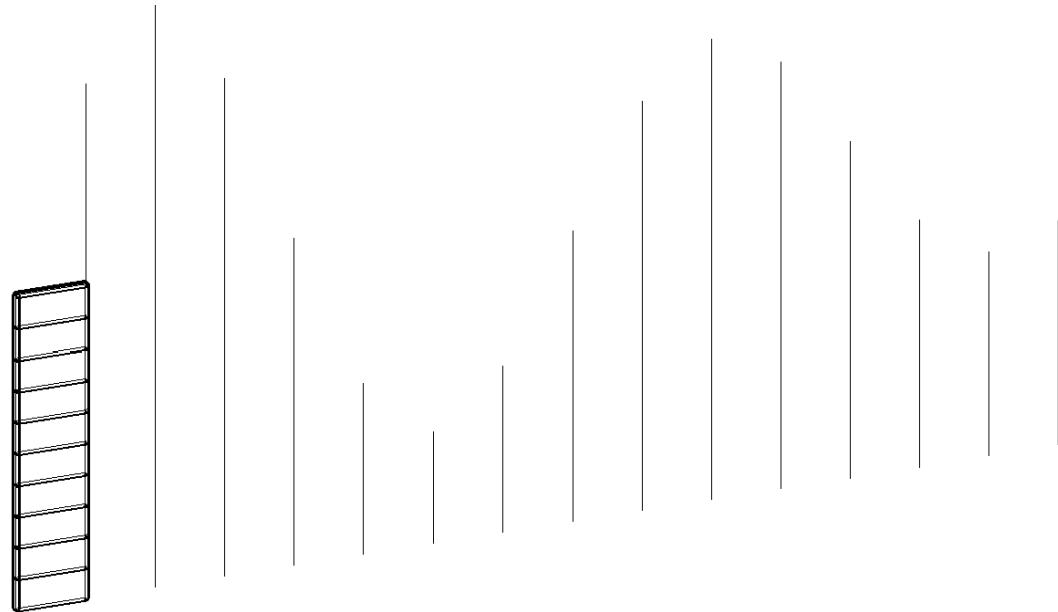
Creates a vertical line of `framed_panels` given a list of points in z. Vector v considers the panel width:

```
vertical_panel_line (generic function with 1 method)
```

```
. vertical_panel_line(pts, v)=
.     let pav_panel(p, q) = framed_panel([p, p+v, q+v, q])
.         [pav_panel(p, q) for (p, q) in zip(pts[1:end-1], pts[2:end])]
.     end
```

```
. #=
. begin
.     backend(autocad)
.     n_y = 40
.     n_z = 10
.     dist_y = pav_length-2*d_length
.     dist_z = pav_height-d_height
.     west_glass_wall = damped_sin_glass_wall(x(d_width),
.                                         amp_y_max_top,
.                                         fi, decay, om_y,
.                                         dist_y, dist_z, n_y, n_z)
.     east_glass_wall = damped_sin_glass_wall(x(pav_width-d_width),
.                                         amp_y_min_bottom,
.                                         fi, decay, om_y,
.                                         dist_y, dist_z, n_y, n_z)
.     panel_width = dist_y/n_y
.     vertical_panel_line(west_glass_wall[1], vy(1), panel_width)
. end
.=#
```

Expected result:



Makes a vertical_panel_line for all lines in the matrix:

```
vertical_panel_lines (generic function with 1 method)
```

```
. vertical_panel_lines(mtx)=
.     let v = mtx[2][1]-mtx[1][1]
```

```
.     [vertical_panel_line(pts, v) for pts in mtx]
.
end
```

Predicate functions: is current z above the limit?

```
is_above_z_limit (generic function with 1 method)
.
is_above_z_limit(z, z_lim)= z > z_lim
```

vert_pts creates the origin points for all vertical panels above p. Stops creating points in z when the set limit is reached:

```
vert_pts (generic function with 1 method)
.
vert_pts(p, panel_height, z_lim) = is_above_z_limit(p.z, z_lim) ? [p] : [p,
vert_pts(p+vz(panel_height), panel_height, z_lim)...]
```

Different starting point for vertical panels in the odd case:

```
vert_pts_odd (generic function with 1 method)
.
vert_pts_odd(p, panel_height, z_lim) = [p, vert_pts(p+vz(panel_height/2), panel_height, z_lim)...]
```

Interweaving 2 different functions depending on count being even or odd:

```
f_weave (generic function with 1 method)
.
f_weave(f1, f2, count) = isodd(count) ? f1 : f2
```

damped_sin_glass_pts creates a vertical line of panels for all points between p and p+vy(dist_y). The z_limit for all vertical lines of panels is defined by the z value of the damped_sin_wave at any moment in the y progression. Interweaving of functions. Different starting height for the first panel of odd rows.

```
damped_sin_glass_pts (generic function with 1 method)
.
damped_sin_glass_pts(p, a_y, fi, decay, om_y, dist_y, dist_z, n_y, n_z, panel_height) =
.
let pts = map_division(y -> p+vy(y), 0, dist_y, n_y)
.
zs = map_division(y -> dist_z+damped_sin_wave(a_y, decay, om_y, y), 0, dist_y, n_y)
.
ns = 1:length(zs)
.
[f_weave(vert_pts_odd(p, panel_height, z_lim), vert_pts(p, panel_height, z_lim), count) for (p,
z_lim, count) in zip(pts, zs, ns)]
.
end
```

glass_panel_height defines façade glass panel's height (width is defined by the number of points in the damped_sinusoid array)

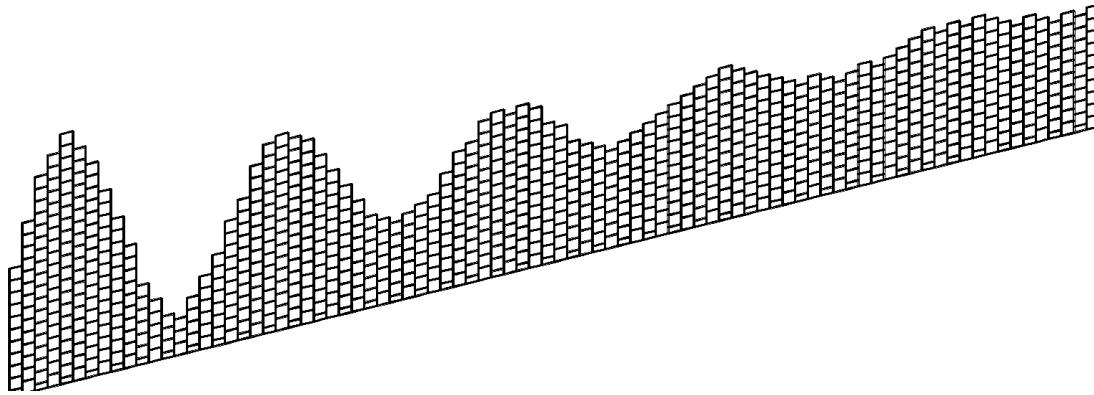
```
.
#=
```

```

. begin
.   backend(autocad)
.   west_glass_wall = damped_sin_glass_pts(x(d_width),
.                                             amp_y_max_top,
.                                             fi, decay, om_y,
.                                             dist_y, dist_z, n_y, n_z, glass_panel_height)
.   east_glass_wall = damped_sin_glass_pts(x(pav_width-d_width),
.                                             amp_y_min_bottom,
.                                             fi, decay, om_y,
.                                             dist_y, dist_z, n_y, n_z, glass_panel_height)
.   delete_all_shapes()
.   vertical_panel_lines(west_glass_wall)
. end
. #=

```

Expected result:



vertical_panel_line corrections to match roof wave:

```

vertical_panel_line_centred (generic function with 1 method)

. vertical_panel_line_centred(pts, v)=
.   let pav_panel(p, q) = framed_panel([p, p+v, q+v, q])
.     [pav_panel(p-v/2, q-v/2) for (p, q) in zip(pts[1:end-1], pts[2:end])]
. end

```

centred_vertical_panel_lines (generic function with 1 method)

```

. centred_vertical_panel_lines(mtx)=
.   let v = mtx[2][1]-mtx[1][1]
.     [vertical_panel_line_centred(pts, v) for pts in mtx]
. end

```

```

. #=
. begin
.   backend(autocad)
.   delete_all_shapes()
.   west_glass_wall_i = damped_sin_glass_wall(xy(d_width, d_length),
.                                             amp_y_max_top,
.                                             fi, decay, om_y_bottom,
.                                             dist_y, dist_z, n_y, n_z, glass_panel_height)

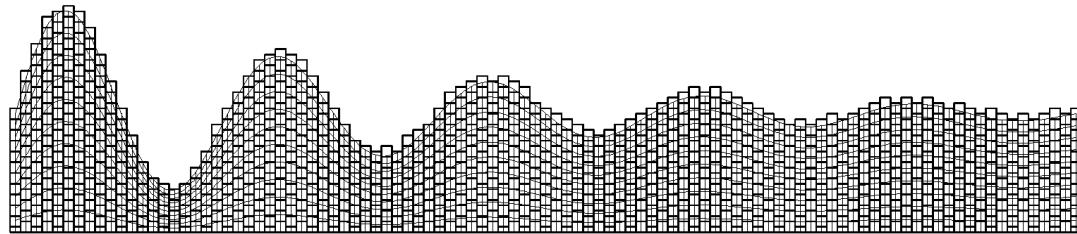
```

```

.
.
.
west_glass_wall_r = damped_sin_glass_pts(xy(d_width, d_length),
                                         dist_y, dist_z, n_y, n_z)
                                         amp_y_max_top,
                                         fi, decay, om_y_bottom,
                                         dist_y, dist_z, n_y, n_z, glass_panel_height)
.
splines4surf(west_glass_wall_i)
centred_vertical_panel_lines(west_glass_wall_r)
end
#=#

```

Expected result:



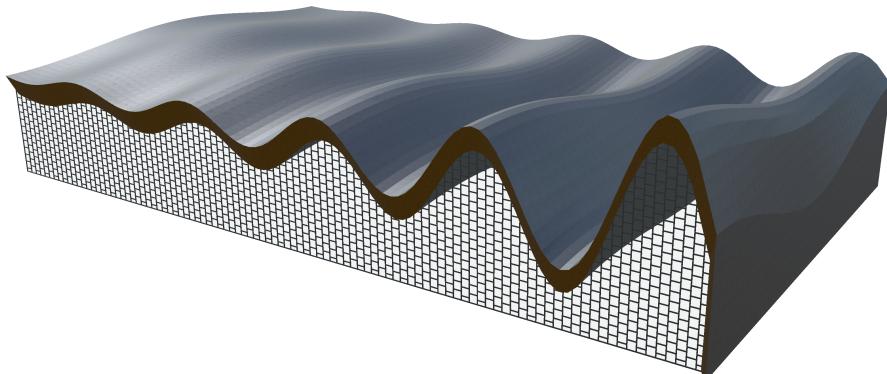
```

.
#=
begin
    backend(unity)
    delete_all_shapes()
    ground()
    roof_surf(uθ(), 50, 100)
    centred_vertical_panel_lines(west_glass_wall)
    centred_vertical_panel_lines(east_glass_wall)
end
#=#

```

```
. # render_view("roof_side_walls_unity")
```

Expected result:



Front glass wall

damped_sin_front_wall (generic function with 1 method)

```

. damped_sin_front_wall(p, a_x, fi, decay, om_x, dist_x, dist_z, n_x, n_z, panel_height) =
.     let pts = map_division(x -> p+vx(x), 0, dist_x, n_x)
.         zs = map_division(x -> dist_z+sin(x/dist_x*pi)*sinusoidal(a_x, om_x, fi-pi, x), 0, dist_x,
n_x)
.         ns = 1:length(zs)
.         [f_weave(vert_pts_odd(p, panel_height, z_lim), vert_pts(p, panel_height, z_lim), count) for (p,
z_lim, count) in zip(pts, zs, ns)]
.     end

```

This function joins all glass walls in the pavilion (sides and front):

pav_walls (generic function with 1 method)

```

. pav_walls(p, n_x, n_y, n_z)=
.     let dist_y = pav_length-2*d_length
.         dist_z = pav_height-d_height
.         dist_x = pav_width-d_width*2
.         west_glass_wall = damped_sin_glass_wall(p+vxy(d_width, d_length),
.                                         amp_y_max_top,
.                                         fi, decay, om_y_bottom,
.                                         dist_y, dist_z, n_y, n_z, glass_panel_height)
.         east_glass_wall = damped_sin_glass_wall(p+vxy(pav_width-d_width, d_length),
.                                         amp_y_min_bottom,
.                                         fi, decay, om_y_bottom,
.                                         dist_y, dist_z, n_y, n_z, glass_panel_height)
.         panel_w = norm(west_glass_wall[2][1]-west_glass_wall[1][1])
.         north_glass_wall = damped_sin_front_wall(p+vxy(d_width, dist_y+panel_w/2),
.                                         amp_x, fi, decay, om_x,
.                                         dist_x, dist_z, n_x, n_z, glass_panel_height)
.         centred_vertical_panel_lines(west_glass_wall[1:end-1])
.         centred_vertical_panel_lines(east_glass_wall[1:end-1])
.         vertical_panel_lines(north_glass_wall[1:end-1])
.     end

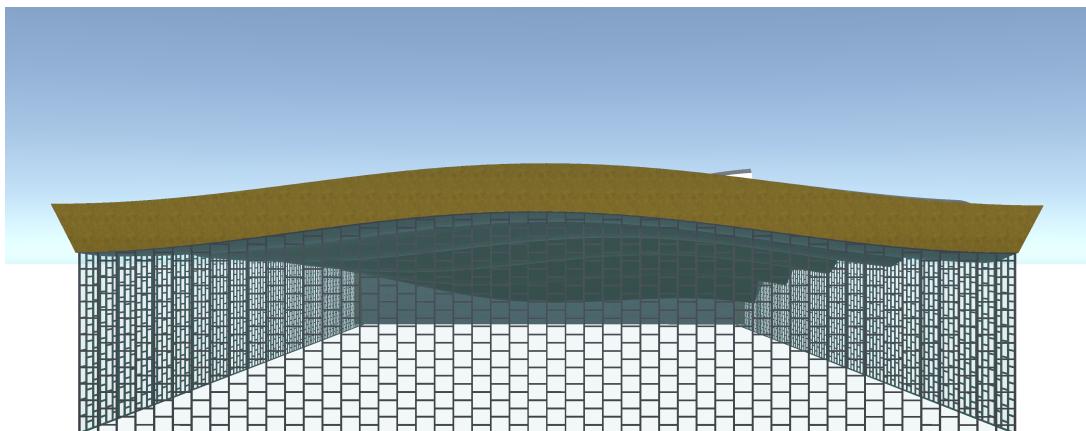
```

```

. #=
. begin
.     backend(unity)
.     delete_all_shapes()
.     ground()
.     roof_surf(u0(), 50, n_panels_y)
.     pav_walls(u0(), 50, n_panels_y, n_glass_verts)
.     set_view(xyz(29.727153778076172, 163.0078582763672, 10.998817443847656),
xyz(29.728837966918945, 162.0078582763672, 10.998811721801758), 35.0)
.     #     render_view("all_walls_unity")
.     end
. #

```

Expected result:



Interior Plan

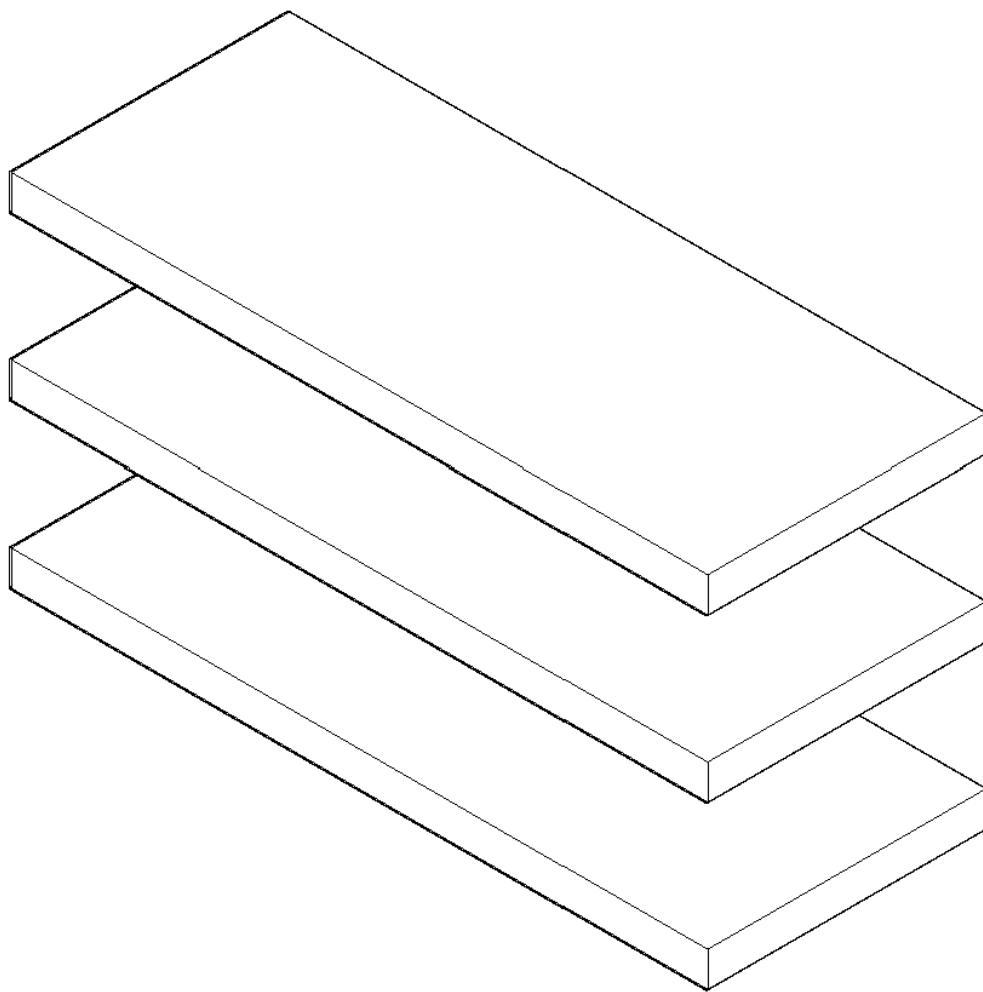
pav_slabs creates the main slabs

pav_slabs (generic function with 1 method)

```
. pav_slabs(p, length, height, width, n_floors)=
.   for i in division(0, height-2, n_floors, false)
.     slab(rectangular_path(p, width, length), level=i, family=pav_slab_fam)
.   end
```

```
. #=
. begin
.   backend(autocad)
.   delete_all_shapes()
.   pav_slabs(u0(), 10, 7, 4, 3)
. end
.=#
```

Expected result:



Update pav_slabs to include interior walls:

```
pav_slabs_walls (generic function with 1 method)
· pav_slabs_walls(p, length, height, width, n_floors, n_x, n_y)=
·     let level_to_level_height = (height-2)/n_floors
·         for i in division(0, height-2, n_floors, false)
·             slab(rectangular_path(p, width, length), level=i, family=pav_slab_fam)
·         end
·         for i in division(0, height-2-level_to_level_height, n_floors-1, false)
·             walls_x = [wall([p+vx(x), p+vxy(x, length)], bottom_level=i,
· top_level=level_to_level_height+i)
·                     for x in division(width/n_x-1, width, n_x, false)]
·             walls_y = [wall([p+vy(y), p+vxy(width, y)], bottom_level=i,
· top_level=level_to_level_height+i)
·                     for y in division(length/n_y-1, length, n_y, false)]
·         end
·     end
```

floors joins structural walls to the previous function:

```
floors (generic function with 1 method)
```

```
function floors(p)
```

```

. let panel_w = (pav_length-2*d_length)/n_panels_y
. dist_x = pav_width-d_width*2
. dist_y_1 = (pav_length-d_length*2)/3
. dist_y_2 = (pav_length-d_length*2)*2/3
. dist_z = pav_height-d_height
.
. # structural walls
. wall([p+vxy(d_width, dist_y_1), p+vxy(pav_width-d_width, dist_y_1)],
.       top_level = level(dist_z), family=pav_wall_struct_fam)
. wall([p+vxy(d_width, dist_y_2), p+vxy(pav_width-d_width, dist_y_2)],
.       top_level = level(dist_z), family=pav_wall_struct_fam)
.
. # floor plans
. pav_slabs_walls(p+vxy(d_width, dist_y_1), dist_y_2+panel_w/2, dist_z,
.                   dist_x, n_floors, n_wall_in_width, n_wall_in_length)
. slab(rectangular_path(p+vx(d_width), dist_x, dist_y_1), level=0, family=pav_slab_fam)
.
. end
. end

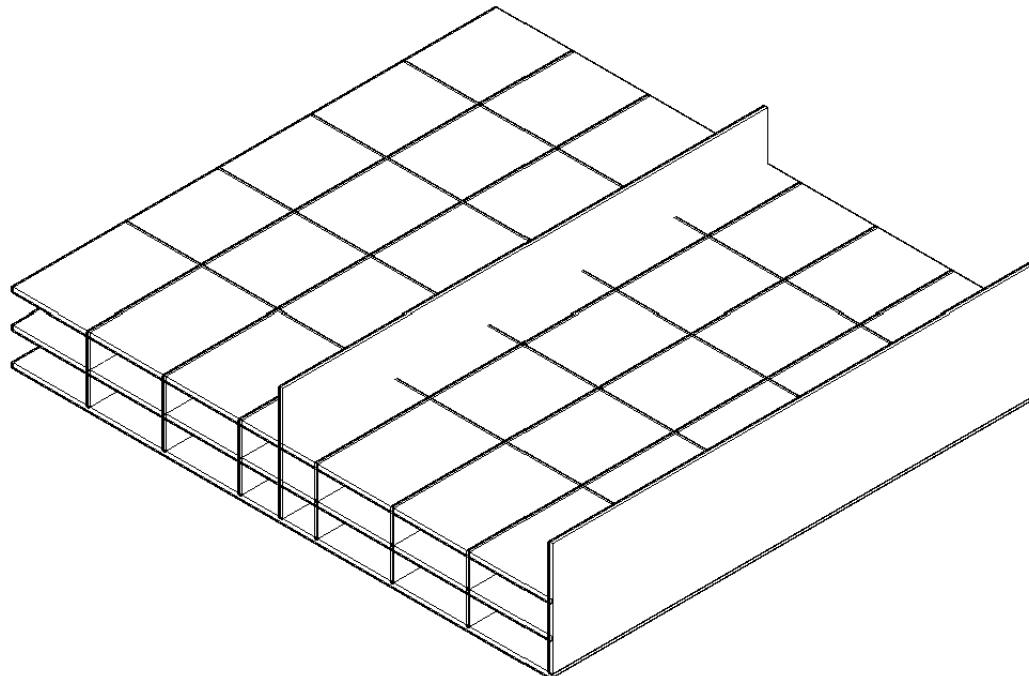
```

```

. #=
. begin
.   backend(autocad)
.   delete_all_shapes()
.   floors(u0())
. end
. =#

```

Expected result:



```
. #=

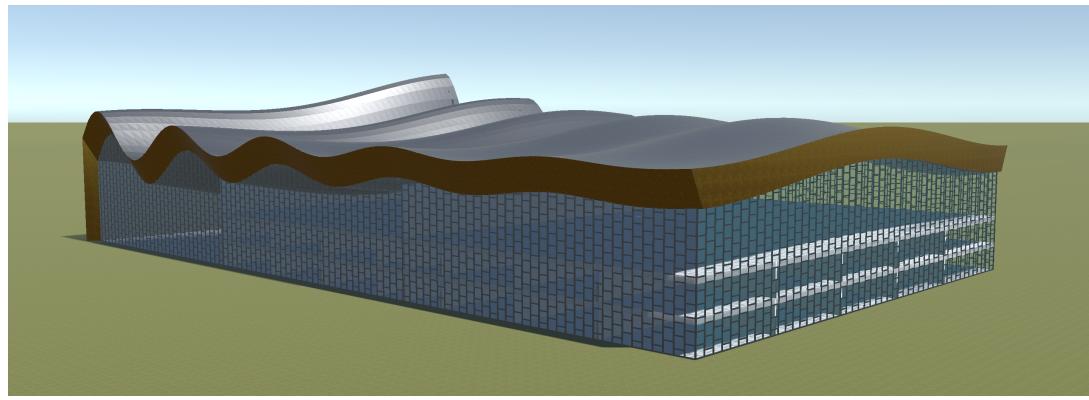
```

```

. begin
.   backend(unity)
.   delete_all_shapes()
.   set_backend_family(ground_fam, unity, unity_material_family("Default/Materials/Grass"))
.   ground()
.   pav_walls(u0(), n_panels_x, n_panels_y, n_glass_verts)
.   roof_surf(u0(), n_panels_x, n_panels_y)
.   floors(u0())
.   set_view(xyz(120.72909545898438, 142.31788635253906, 18.540668487548828),
xyz(120.00056457519531, 141.63906860351562, 18.4487361907959), 35.0)
. #     render_view("interior_walls_unity")
. end
. =#

```

Expected result:



Site

Ground:

ground (generic function with 1 method)

```

. ground() =
.   let x = 1000
.   y = 1000
.   z = -0.01
.   slab(closed_polygonal_path([xyz(-x,y,z), xyz(x,y,z), xyz(x,-y,z), xyz(-x,-y,z)]), level(-0.05),
ground_fam)
. end

```

Random tree definition:

leaf (generic function with 1 method)

```

. leaf(p, r, mat) =
.   if current_backend() == unity
.     with(current_material, get_material(mat)) do
.       sphere(p, r)
.     end

```

```

.   else
.     sphere(p, r)
.   end

```

branch (generic function with 1 method)

```

. branch(pb, rb, pt, rt, mat)=
.   if current_backend() == unity
.     with(current_material, get_material(mat)) do
.       cone_frustum(pb, rb, pt, rt)
.     end
.   else
.     cone_frustum(pb, rb, pt, rt)
.   end

```

tree (generic function with 1 method)

```

. tree(p_base, r_base, w_branch, fi, psi, min_f_w, max_f_w, max_fi, min_psi, max_psi) =
.   let psi = psi > pi/2 ? psi - pi/2 : psi < -pi/2 ? psi + pi/2 : psi
.     p_top = p_base + vsph(w_branch, fi, psi)
.     r_top = r_base/2
.     leaf_mat = "Default/Materials/Grass"
.     branch_mat = "materials/wood/ExteriorWood9"
.     branch(p_base, r_base, p_top, r_top, branch_mat)
.     if w_branch < 0.01 || r_top < 0.001
.       leaf(p_top, 0.05, leaf_mat)
.     else
.       tree(p_top, r_top, w_branch*random_range(min_f_w, max_f_w),
.             fi + random(max_fi), psi + random_range(min_psi, max_psi),
.             min_f_w, max_f_w, max_fi, min_psi, max_psi)
.       tree(p_top, r_top, w_branch*random_range(min_f_w, max_f_w),
.             fi - random(max_fi), psi - random_range(min_psi, max_psi),
.             min_f_w, max_f_w, max_fi, min_psi, max_psi)
.     end
.   end

```

random_tree (generic function with 1 method)

```

. random_tree(p)=
.   let r_base = random_range(0.05, 0.3)
.     w_branch = random_range(2, 5)
.     fi = 0
.     psi = 0
.     min_f_w = 0.6 # branch width max factor
.     max_f_w = 0.9 # branch width min factor
.     max_fi = 2pi
.     min_psi = pi/16
.     max_psi = pi/2
.     tree(p, r_base, w_branch, fi, psi, min_f_w, max_f_w, max_fi, min_psi, max_psi)
.
.   end

```

```

. #
. begin

```

```

. backend(unity)
. delete_all_shapes()
. ground()
. map(n -> random_tree(x(n)), 0:2:20)
. #     render_view("tress")
. end
.=#

```

Expected result:



Complete Building

The GymPav function gathers all building elements modeled, plus the random tree line in front of the west façade. Given a point p, the function creates the full building structure in accordance with the parameters provided in Pavilion Dimensions.

GymPav (generic function with 1 method)

```

. function GymPav(p)
.     roof_surf(p, n_panels_x, n_panels_y)
. #     roof_truss(p, 30, 70)
.     pav_walls(p, n_panels_x, n_panels_y, n_glass_verts)
.     floors(p)
.     map(n -> random_tree(p+vxy(-5, n)), 0:2:pav_length) # tree line in the west façade
. end

```

```

. #=
. begin
.     backend(unity)
.     delete_all_shapes()
.     ground()
.     GymPav(u0())
. end
.=#

```

Saved render views:

```

. begin
.   # set_view(xyz(-27.78546905517578, -12.493826866149902, 2.73970365524292),
xyz(-27.147165298461914, -11.735978126525879, 2.874735116958618), 35.0)
.     # render_view("pav1")
.

.   # set_view(xyz(37.13166809082031, 96.70274353027344, 8.049731254577637),
xyz(36.132999420166016, 96.71927642822266, 8.098581314086914), 35.0)
.     # render_view("pav2")
.

.   # set_view(xyz(86.96595001220703, 87.33833312988281, 7.179433822631836),
xyz(85.97071075439453, 87.31035614013672, 7.27283239364624), 35.0)
.     # render_view("pav3")
.

.   # set_view(xyz(-22.71254539489746, 144.94818115234375, 7.690451622009277),
xyz(-22.17136001586914, 144.1073455810547, 7.7015814781188965), 35.0)
.     # render_view("pav4")
.

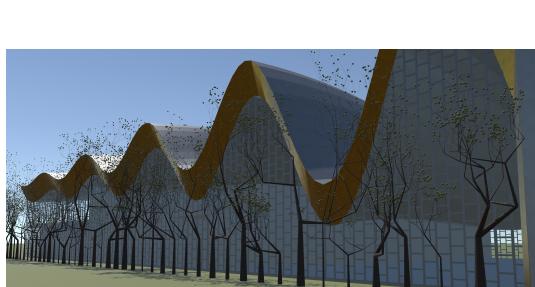
.   # set_view(xyz(-12.342000961303711, 103.7385025024414, 3.020662307739258),
xyz(-11.917301177978516, 102.83597564697266, 3.0918772220611572), 35.0)
.     # render_view("pav5")
.

.   # set_view(xyz(-2.6654765605926514, 130.2740020751953, 1.4012055397033691),
xyz(-2.5774435997009277, 129.29388427734375, 1.5790280103683472), 35.0)
.     # render_view("pav6")
.

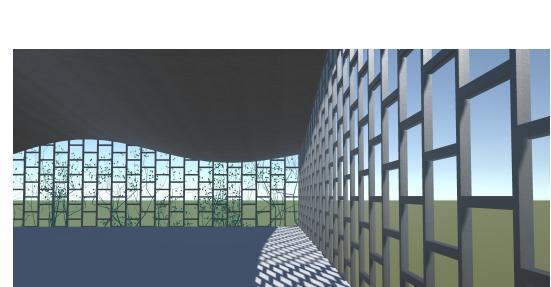
.   # set_view(xyz(126.3520736694336, -49.96141815185547, 27.672582626342773),
xyz(125.61244201660156, -49.31004333496094, 27.503311157226562), 35.0)
.     # render_view("pav7")
.
end

```

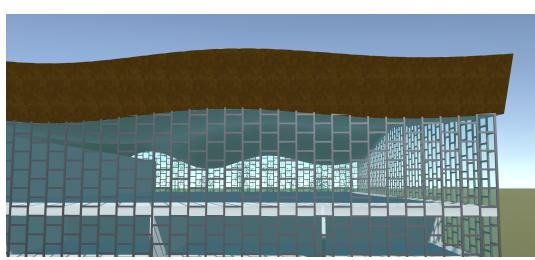
Unity renders



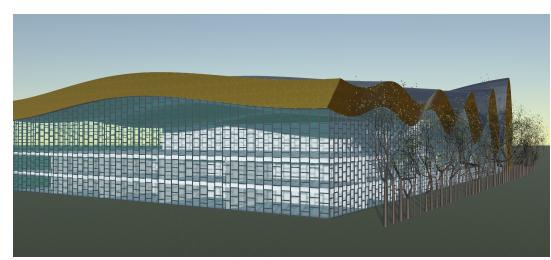
Pav 1



Pav 2



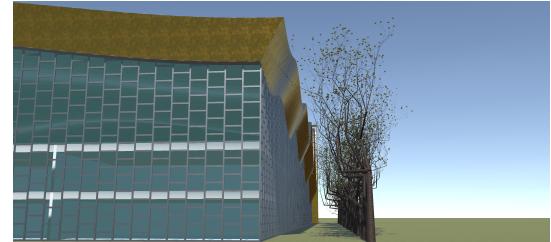
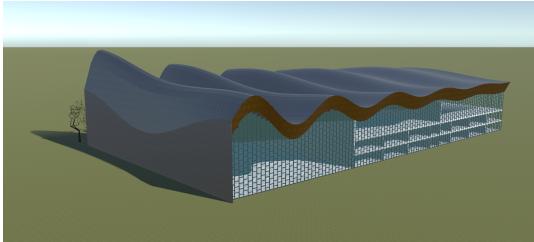
Pav 3



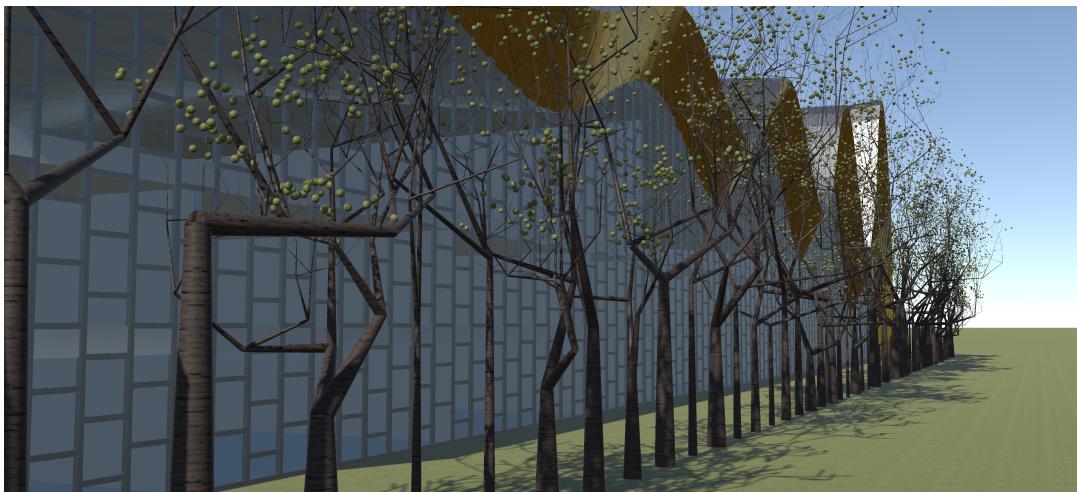
Pav 4

Pav 7

Pav 6



Pav5



The end