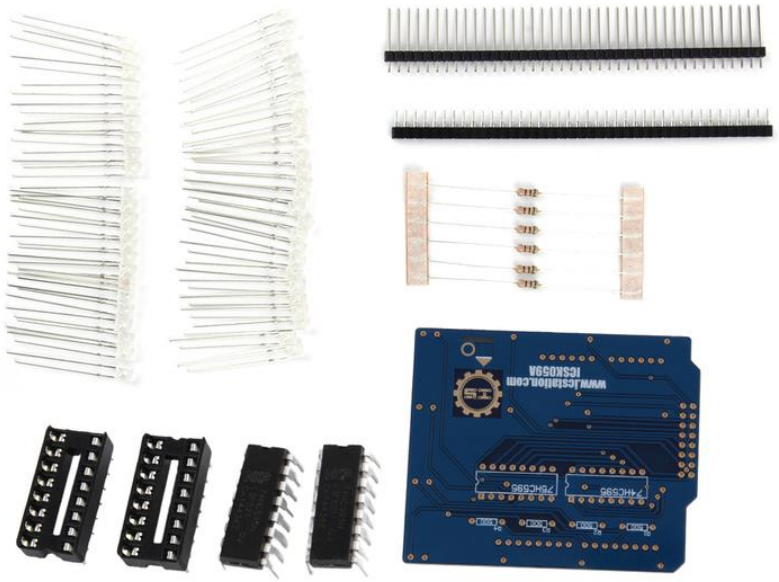
1456082837
VÝROBNÍ ČÍSLO

Arduino 4x4x4 3D LED kostka shield

1. POPIS

Shield obsahuje celkem 64 LED diod v uspořádání 4x4x4. Po sestavení stavebnice diody tvoří trojrozměrný objekt – kostku. Každou diodu je možné spínat nezávisle na ostatních díky posuvným registrům 74HC595.

Shield je kompatibilní s vývojovými kity Arduino UNO, Duemilanove, Diecimila, ADK/Mega R3 a vyššími, přes které je také napájen.



Obsah balení:

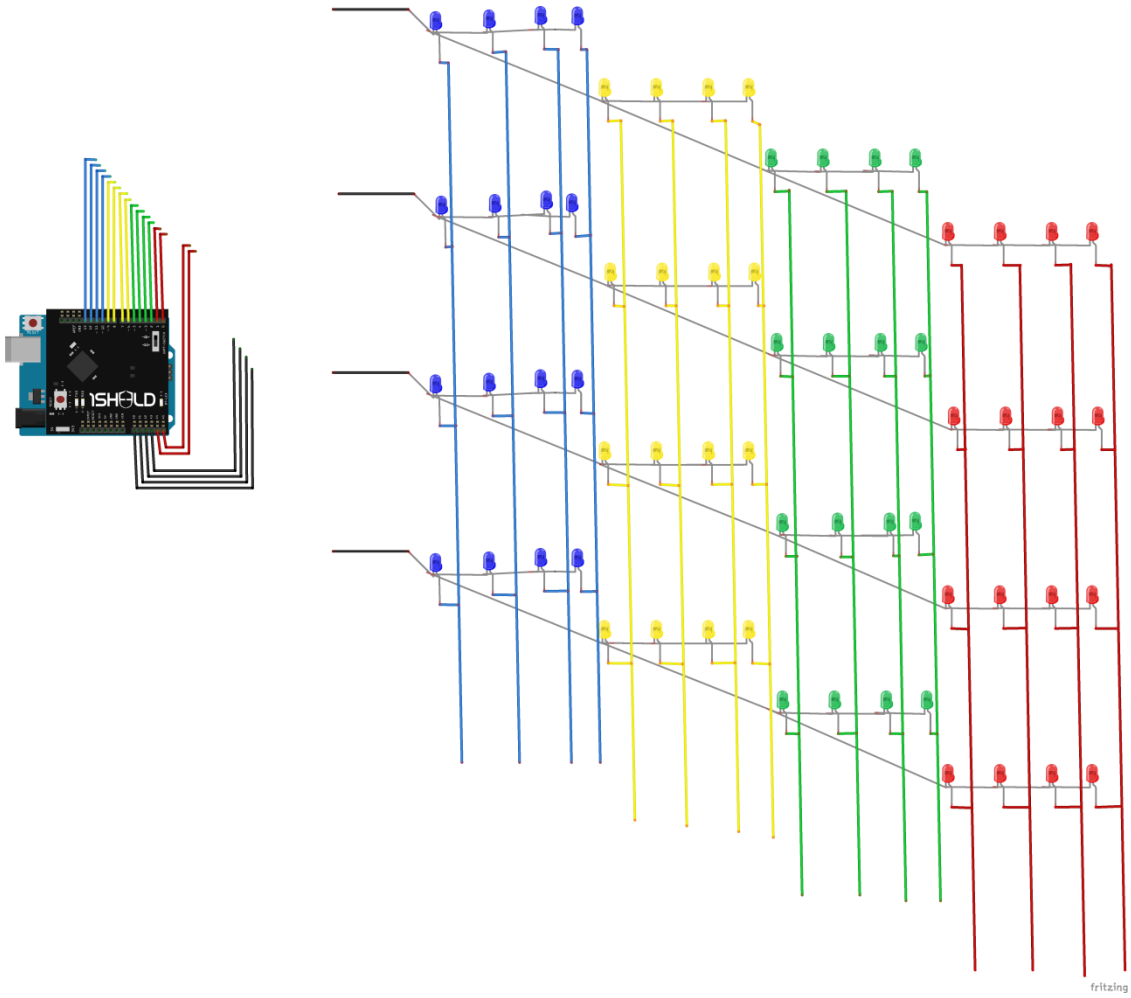
- 64x LED diody (modré)
- 1x plošný spoj
- 2x pinový pás
- 6x rezistor 500 Ω
- 2x posuvný registr 74HC595 + patice

2. ZAPOJENÍ

Pro zprovoznění spájeného shieldu nejsou zapotřebí žádné externí komponenty, stačí jej vložit přímo do patice vývojového kitu Arduino.

LED diody spájejte dle ilustračního obrázku níže.

Upozornění: Při pájení si dávejte pozor na polaritu LED diod. Otočení polarity má za následek nesprávnou funkci shieldu, nemělo by však dojít k poškození shieldu nebo vývojového kitu. Ilustrační obrázek neobsahuje rezistory – zapojte vždy jeden rezistor sériově mezi základní desku shieldu a řádek kostky (mezi černé linky).



Převzato z: https://create.arduino.cc/projecthub/ihassanibrahim7/voice-controlled-led-cube-with-arduino-uno-and-1sheeld-69afc2

3. UKÁZKA PROGRAMU

Ukázka níže je převzata z ukázkového kódu knihovny „MD_Cubo“, která je ke stažení zde: https://arduinocode.codeplex.com/releases/view/616782

```
// LED Cube demo pattens
//
// Some original and some shamelessly copied from YouTube video of other cubes
//

#include <MD_Cubo.h>
#include "MD_Cubo_Font.h"

// Pick the include file for the right cube
//#include "MD Cubo 4x4 72xx.h"
//#include "MD_Cubo_4x4_ICS595.h"
#include "MD_Cubo_8x8_jC.h"

// Define the cube object
#ifdef MD CUBO 4x4 72XX H
MD_Cubo_72xx  C;
#endif
```

```
#ifdef MD_CUBO_4x4_ICS595_H
MD Cubo ICS595  C;
#endif
#ifdef MD CUBO 8x8 JC H
MD_Cubo_JC         C;
#endif

#define RANDOM CYCLE  1 // 1 for random selection, 0 for sequential cycle

#define   DEBUG      0                // Enable or disable (default) debugging output from the example

#if DEBUG
#define   PRINT(s, v)                 { Serial.print(F(s)); Serial.print(v); }                 // Print a
string followed by a value (decimal)
#define   PRINTX(s, v)       { Serial.print(F(s)); Serial.print(v, HEX); }   // Print a string followed by
a value (hex)
#define   PRINTB(s, v)       { Serial.print(F(s)); Serial.print(v, BIN); }   // Print a string followed by
a value (binary)
#define   PRINTS(s)            { Serial.print(F(s)); }
         // Print a string
#define PRINTC(s, x, y, z) { PRINTS(s); PRINT("(",x); PRINT(",",y); PRINT(",",z); PRINTS(")"); }  // Print
coordinate tuple
#else
#define   PRINT(s, v)                 ///< Print a string followed by a value (decimal)
#define   PRINTX(s, v)       ///< Print a string followed by a value (hex)
#define   PRINTB(s, v)       ///< Print a string followed by a value (binary)
#define   PRINTS(s)          ///< Print a string
#define PRINTC(s, x, y, z)  ///< Print coordinate tuple
#endif

// Most demos run for a DEMO RUNTIME ms before ending.
// De4fine constants and time tracking variable
#define DEMO_RUNTIME  20000L
uint32 t timeStart;

void firefly()
// LEDs on and off like fireflies in the night
{
  uint8_t x, y, z;

  PRINTS("\nFirefly");
  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    C.clear();
    for (uint8_t x = 0; x<C.size(MD_Cubo::XAXIS); x++)
      for (uint8_t y = 0; y<C.size(MD_Cubo::YAXIS); y++)
        for (uint8_t z = 0; z<C.size(MD_Cubo::ZAXIS); z++)
          C.setVoxel(random(100)>50, x, y, z);
    C.update();
    C.animate(50);
  }
}

void brownian()
// 2x2 cube doing brownian motion
{
  uint8 t x = 0, y = 0, z = 0;
  int8 t dx, dy, dz;

  PRINTS("\nBrownian");
  timeStart = millis();
  while (millis() - timeStart <= DEMO RUNTIME)
  {
    C.clear();
    // PRINTC("\n", x, y, z);
    C.drawCube(true, x, y, z, 2);
    C.update();
    C.animate(75);

    dx = random(3) - 1;
    dy = random(3) - 1;
    dz = random(3) - 1;
```

```
    if ((x + dx >= 0) && (x + dx + 1 < C.size(MD_Cubo::XAXIS))) x += dx;
    if ((y + dy >= 0) && (y + dy + 1 < C.size(MD_Cubo::YAXIS))) y += dy;
    if ((z + dz >= 0) && (z + dz + 1 < C.size(MD_Cubo::ZAXIS))) z += dz;
  }
}

void slideFaces()
// Slide the faces around the horizontal center of the cube
{
  const uint16_t  delay = 100;

  PRINTS("\nSlide Faces");

  C.clear();
  C.fillPlane(true, MD_Cubo::XYPLANE, 0);
  C.update();
  C.animate(delay);

  timeStart = millis();
  while (millis() - timeStart <= DEMO RUNTIME)
  {
    for (uint8_t j = 0; j<C.size(MD_Cubo::YAXIS) - 1; j++)
    {
      C.drawLine(false, 0, j, 0, C.size(MD_Cubo::XAXIS) - 1, j, 0);
      C.drawLine(true, 0, C.size(MD_Cubo::YAXIS) - 1, j + 1, C.size(MD_Cubo::XAXIS) - 1,
C.size(MD_Cubo::YAXIS) - 1, j + 1);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j = 0; j<C.size(MD_Cubo::ZAXIS) - 1; j++)
    {
      C.drawLine(false, 0, C.size(MD_Cubo::YAXIS) - 1, j, C.size(MD_Cubo::XAXIS) - 1,
C.size(MD_Cubo::YAXIS) - 1, j);
      C.drawLine(true, 0, C.size(MD_Cubo::YAXIS) - 2 - j, C.size(MD_Cubo::ZAXIS) - 1,
C.size(MD_Cubo::XAXIS) - 1, C.size(MD_Cubo::YAXIS) - 2 - j, C.size(MD_Cubo::ZAXIS) - 1);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j = 0; j<C.size(MD_Cubo::YAXIS) - 1; j++)
    {
      C.drawLine(false, 0, C.size(MD_Cubo::YAXIS) - 1 - j, C.size(MD_Cubo::ZAXIS) - 1,
C.size(MD_Cubo::XAXIS) - 1, C.size(MD_Cubo::YAXIS) - 1 - j, C.size(MD_Cubo::ZAXIS) - 1);
      C.drawLine(true, 0, 0, C.size(MD_Cubo::ZAXIS) - 2 - j, C.size(MD_Cubo::XAXIS) - 1, 0,
C.size(MD_Cubo::ZAXIS) - 2 - j);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j = 0; j<C.size(MD_Cubo::ZAXIS) - 1; j++)
    {
      C.drawLine(false, 0, 0, C.size(MD_Cubo::ZAXIS) - 1 - j, C.size(MD_Cubo::XAXIS) - 1, 0,
C.size(MD_Cubo::ZAXIS) - 1 - j);
      C.drawLine(true, 0, j + 1, 0, C.size(MD_Cubo::XAXIS) - 1, j + 1, 0);
      C.update();
      C.animate(delay);
    }
  }
}

void wrapFaces()
// Wrap the around all the 6 faces of the cube in a sliding pattern
{
  const uint16_t  delay = 100;

  PRINTS("\nWrap Faces");

  C.clear();
  C.fillPlane(true, MD_Cubo::XYPLANE, 0);
  C.update();
  C.animate(delay);

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
```

```
    for (uint8_t j=0; j<C.size(MD_Cubo::YAXIS)-1; j++)
    { // move across the XY plane @ Z 0 into XZ plane @ Y max
      C.drawLine(false, 0, j, 0, C.size(MD_Cubo::ZAXIS)-1, j, 0);
      C.drawLine(true, 0, C.size(MD_Cubo::YAXIS)-1, j+1, C.size(MD_Cubo::XAXIS)-1, C.size(MD_Cubo::YAXIS)-
1, j+1);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j=0; j<C.size(MD_Cubo::ZAXIS)-1; j++)
    { // move across the XZ plane @ Y max into XY plane @ Z max
      C.drawLine(false, 0, C.size(MD_Cubo::YAXIS)-1, j, C.size(MD_Cubo::XAXIS)-1, C.size(MD_Cubo::YAXIS)-
1, j);
      C.drawLine(true, 0, C.size(MD_Cubo::YAXIS)-2-j, C.size(MD_Cubo::ZAXIS)-1, C.size(MD_Cubo::XAXIS)-1,
C.size(MD_Cubo::YAXIS)-2-j, C.size(MD_Cubo::ZAXIS)-1);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j=0; j<C.size(MD_Cubo::XAXIS)-1; j++)
    { // move across the XY plane @ Z max into YZ plane @ X max
      C.drawLine(false, j, 0, C.size(MD_Cubo::ZAXIS)-1, j, C.size(MD_Cubo::YAXIS)-1,
C.size(MD_Cubo::ZAXIS)-1);
      C.drawLine(true, C.size(MD_Cubo::XAXIS)-1, 0, C.size(MD_Cubo::ZAXIS)-2-j, C.size(MD_Cubo::XAXIS)-1,
C.size(MD_Cubo::YAXIS)-1, C.size(MD_Cubo::ZAXIS)-2-j);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j=0; j<C.size(MD_Cubo::YAXIS)-1; j++)
    { // move across the YZ plane @ X max into XZ plane @ Y 0
      C.drawLine(false, C.size(MD_Cubo::XAXIS)-1, C.size(MD_Cubo::YAXIS)-1-j, 0, C.size(MD_Cubo::XAXIS)-1,
C.size(MD_Cubo::YAXIS)-1-j, C.size(MD_Cubo::ZAXIS)-1);
      C.drawLine(true, C.size(MD_Cubo::XAXIS)-2-j, 0, 0, C.size(MD_Cubo::XAXIS)-2-j, 0,
C.size(MD_Cubo::ZAXIS)-1);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j=0; j<C.size(MD_Cubo::XAXIS)-1; j++)
    { // move across XZ Plane @ Y 0 into YZ plane @ X 0
      C.drawLine(false, C.size(MD_Cubo::XAXIS)-1-j, 0, 0, C.size(MD_Cubo::XAXIS)-1-j, 0,
C.size(MD_Cubo::ZAXIS)-1);
      C.drawLine(true, 0, j+1, 0, 0, j+1, C.size(MD_Cubo::ZAXIS)-1);
      C.update();
      C.animate(delay);
    }
    for (uint8_t j=0; j<C.size(MD_Cubo::ZAXIS)-1; j++)
    { // move across YZ plane @ X 0 into XY plane @ Z 0 - back to starting point
      C.drawLine(false, 0, 0, C.size(MD_Cubo::ZAXIS)-1-j, 0, C.size(MD_Cubo::YAXIS)-1,
C.size(MD_Cubo::ZAXIS)-1-j);
      C.drawLine(true, j+1, 0, 0, j+1, C.size(MD_Cubo::YAXIS)-1, 0);
      C.update();
      C.animate(delay);
    }
  }
}

void intersectPlanes()
// Vertical and horizontal intersecting planes moving end-to-end in unison
{
  const uint16_t  delay = 200;

  PRINTS("\nIntersect Planes");

  C.clear();
  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    for (uint8_t p = 0; p<C.size(MD_Cubo::YAXIS); p++)
    { // move plane one way ...
      C.fillPlane(true, MD_Cubo::XYPLANE, p);
      C.fillPlane(true, MD_Cubo::YZPLANE, p);
      C.update();
      C.animate(delay);
      C.fillPlane(false, MD_Cubo::XYPLANE, p);
      C.fillPlane(false, MD_Cubo::YZPLANE, p);
```

```
    }
    for (uint8_t p = C.size(MD_Cubo::YAXIS) - 2; p>0; --p)
    { // ... reverse back
      C.fillPlane(true, MD_Cubo::XYPLANE, p);
      C.fillPlane(true, MD_Cubo::YZPLANE, p);
      C.update();
      C.animate(delay);
      C.fillPlane(false, MD_Cubo::XYPLANE, p);
      C.fillPlane(false, MD_Cubo::YZPLANE, p);
    }
  }
}

void droplets()
// dropls mobving up and down between the top and bottom
{
  const uint16_t delay = 50;
  const uint16_t MAX_DROP = (C.size(MD_Cubo::XAXIS)*C.size(MD_Cubo::YAXIS)) / 2;
  uint8 t *dropx = (uint8 t *)malloc(MAX DROP);
  uint8 t *dropy = (uint8 t *)malloc(MAX DROP);
  uint8_t get = 0, put = 1;

  PRINTS("\nDroplets");
  memset(dropx, 0, MAX_DROP*sizeof(dropx[0]));
  memset(dropy, 0, MAX DROP*sizeof(dropy[0]));
  C.clear();

  C.fillPlane(true, MD_Cubo::XYPLANE, C.size(MD_Cubo::ZAXIS) - 1);

  timeStart = millis();
  while (millis() - timeStart <= DEMO RUNTIME)
  {
    boolean found;
    uint8 t x, y;

    // find a coordinate to push down
    do
    {
      x = random(C.size(MD_Cubo::XAXIS));
      y = random(C.size(MD_Cubo::YAXIS));

      //PRINT("\nNew drop ", x);
      //PRINT(",", y);
      found = false;
      for (uint16_t i=0; i<MAX_DROP && !found; i++)
        found |= (x == dropx[i]) && (y == dropy[i]);
      //PRINT("- search ", found);
    } while (found);

    if (put == get)   // raise the oldest one
    {
      //PRINTS(" - overwrite");
      for (uint8_t z = 0; z<C.size(MD_Cubo::ZAXIS) - 1; z++)
      {
        C.setVoxel(false, dropx[get], dropy[get], z);
        C.setVoxel(true, dropx[get], dropy[get], z+1);
        C.update();
        C.animate(delay);
      }
      get = (get+1) % MAX_DROP;
    }

    dropx[put] = x;
    dropy[put] = y;
    put = (put+1) % MAX_DROP;

    // drop the new one
    for (uint8_t z = C.size(MD_Cubo::ZAXIS) - 1; z>0; --z)
    {
      C.setVoxel(false, x, y, z);
      C.setVoxel(true, x, y, z-1);
      C.update();
      C.animate(delay);
```

```
    }
  }

  // now finish up by returning all the drops to the top
  //PRINTS("\nFinishing up");
  do
  {
    for (uint8_t z = 0; z<C.size(MD_Cubo::ZAXIS) - 1; z++)
    {
      C.setVoxel(false, dropx[get], dropy[get], z);
      C.setVoxel(true, dropx[get], dropy[get], z+1);
      C.update();
      C.animate(delay);
    }
    get = (get+1) % MAX_DROP;
  } while (get != put);

  free(dropx);
  free(dropy);
}

void scaleCube()
// Scale a cube from the largest size to the smallest in the center
{
  const uint16_t delay = 125;

  C.clear();
  for (uint8_t i = 0; i < C.size(MD_Cubo::ZAXIS) / 2; i++)
  {
    C.drawCube(true, i, i, i, C.size(MD_Cubo::ZAXIS) - (2 * i));
    C.update();
    C.animate(delay);
    C.drawCube(false, i, i, i, C.size(MD_Cubo::ZAXIS) - (2 * i));
  }

  for (uint8_t i = 1; i <= C.size(MD_Cubo::ZAXIS) / 2; i++)
  {
    C.drawCube(true, (C.size(MD_Cubo::ZAXIS) / 2) - i, (C.size(MD_Cubo::ZAXIS) / 2) - i,
(C.size(MD_Cubo::ZAXIS) / 2) - i, 2 * i);
    C.update();
    C.animate(delay);
    C.drawCube(false, (C.size(MD_Cubo::ZAXIS) / 2) - i, (C.size(MD_Cubo::ZAXIS) / 2) - i,
(C.size(MD_Cubo::ZAXIS) / 2) - i, 2 * i);
  }
}

void shrinkCube()
// Shrink a cube from largest to smallest in the corner.
// Random corners picked for each turn
{
  const uint16_t delay = 75;
  uint8_t cur = 0;
  uint8_t x, y, z, dx, dy, dz;
  int8_t corners[8][6] = // corner coordinates and delta multipliers for size
  {
    { 0, 0, 0 , 1, 1, 1 },
    { C.size(MD_Cubo::XAXIS)-1, 0, 0 , -1, 1, 1 },
    { 0, C.size(MD_Cubo::YAXIS)-1, 0, 1, -1, 1 },
    { 0, 0, C.size(MD_Cubo::ZAXIS)-1, 1, 1, -1 },
    { C.size(MD_Cubo::XAXIS)-1, C.size(MD_Cubo::YAXIS)-1, 0, -1, -1, 1 },
    { C.size(MD_Cubo::XAXIS)-1, 0, C.size(MD_Cubo::ZAXIS)-1, -1, 1, -1 },
    { 0, C.size(MD_Cubo::YAXIS)-1, C.size(MD_Cubo::ZAXIS)-1, 1, -1, -1 },
    { C.size(MD_Cubo::XAXIS)-1, C.size(MD_Cubo::YAXIS)-1, C.size(MD_Cubo::ZAXIS)-1, -1, -1, -1 }
  };

  PRINTS("\nShrink Cube");

  C.clear();

  // Grown the cube from max size to min size then back to max,
  // selected another corner and repeat
  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
```

```
  {
    x = corners[cur][0];
    y = corners[cur][1];
    z = corners[cur][2];

    // growth phase
    for (uint8_t i = 1; i < C.size(MD_Cubo::ZAXIS); i++)
    {
      dx = i * corners[cur][3];
      dy = i * corners[cur][4];
      dz = i * corners[cur][5];

      //PRINTC("\nG Cube ", x, y, z);
      //PRINT(" size ", i+1);

      C.drawRPrism(true, x, y, z, dx, dy, dz);
      C.update();
      C.animate(delay);
      C.drawRPrism(false, x, y, z, dx, dy, dz);
    }

    // see if we should throw in a WOW!
    if (random(50) >= 25) scaleCube();

    // pick another corner
    cur = random(ARRAY SIZE(corners));
    x = corners[cur][0];
    y = corners[cur][1];
    z = corners[cur][2];

    // shrink down to this new corner
    for (uint8_t i = C.size(MD_Cubo::ZAXIS)-1; i >= 1; i--)
    {
      dx = i * corners[cur][3];
      dy = i * corners[cur][4];
      dz = i * corners[cur][5];

      //PRINTC("\nS Cube ", x, y, z);
      //PRINT(" size ", i+1);

      C.drawRPrism(true, x, y, z, dx, dy, dz);
      C.update();
      C.animate(delay);
      C.drawRPrism(false, x, y, z, dx, dy, dz);
    }

  }
}

void rain()
// Rain drops passing through the cube, top to bottom
{
  uint16_t delay = 100;

  PRINTS("\nRain");
  C.clear();

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    uint8_t num_drops = random(100) % (C.size(MD_Cubo::YAXIS)-2);

    C.fillPlane(false, MD_Cubo::XYPLANE, C.size(MD_Cubo::ZAXIS)-1);
    for (uint8_t i = 0; i < num_drops; i++)
    {
      uint8_t x = random(C.size(MD_Cubo::XAXIS));
      uint8_t y = random(C.size(MD_Cubo::YAXIS));
      C.setVoxel(true, x, y, C.size(MD_Cubo::ZAXIS) - 1);
    }
    C.update();
    C.animate(delay);

    // move all the Z planes down
```

```
    for (uint8_t i = 1; i < C.size(MD_Cubo::ZAXIS); i++)
      C.copyPlane(MD_Cubo::XYPLANE, i, i-1);
  }
  return;
}

void randomFill()
// Completely fill the cube with a random pattern until all LEDs lit
{
  int c = 0;

  PRINTS("\nRandom fill");
  C.clear();

  while (c < (C.size(MD_Cubo::XAXIS)*C.size(MD_Cubo::YAXIS)*C.size(MD_Cubo::ZAXIS))-1)
  {
    uint8_t x = random(C.size(MD_Cubo::XAXIS));
    uint8_t y = random(C.size(MD_Cubo::YAXIS));
    uint8_t z = random(C.size(MD_Cubo::XAXIS));

    if (C.getVoxel(x, y, z))
      continue;

    C.setVoxel(true, x, y, z);
    C.update();
    C.animate(75);
    c++;
  }
}

float length(int8_t x1, int8_t y1, uint8_t z1, uint8_t x2, uint8_t y2, uint8_t z2)
// Work out the length of a line
{
  int8_t dx = (x1 - x2), dy = (y1 - y2), dz = (z1 - z2);

  return(sqrt((dx*dx) + (dy*dy) + (dz*dz)));
}

void ripples()
// Display a sine wave running out from the center of the cube
// A symmetrical display, so only need to calculate one quarter of the points
{
  const float waveInterval = 1.5;
  uint16_t iteration = 0;
  uint8_t midX = C.size(MD_Cubo::XAXIS) / 2;
  uint8_t midY = C.size(MD_Cubo::YAXIS) / 2;

  // this only works for larger cubes, soreturn if the cube is not big enough
  if ((C.size(MD_Cubo::XAXIS) < 8 || C.size(MD_Cubo::YAXIS) < 8 || C.size(MD_Cubo::ZAXIS) < 8))
    return;

  PRINTS("\nRipples");

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    C.clear();

    for (uint8_t x = 0; x < midX; x++)
    {
      for (uint8_t y = 0; y < midY; y++)
      {
        float dist = length((C.size(MD_Cubo::ZAXIS) - 1) / 2, (C.size(MD_Cubo::YAXIS) - 1) / 2, 0, x, y,
0) / 9.899495 * C.size(MD_Cubo::ZAXIS);
        float height = (C.size(MD_Cubo::ZAXIS) / 2) + (sin((dist / waveInterval) + (float)iteration / 200)
* (C.size(MD_Cubo::ZAXIS) / 2));

        C.setVoxel(true, x, y, (int)height);
        C.setVoxel(true, C.size(MD_Cubo::XAXIS)-1-x, y, (int)height);
        C.setVoxel(true, x, C.size(MD_Cubo::YAXIS)-1-y, (int)height);
        C.setVoxel(true, C.size(MD_Cubo::XAXIS)-1-x, C.size(MD_Cubo::YAXIS)-1-y, (int)height);
        iteration++;
      }
```

```
    }
    C.update();
    C.animate(0);
  }
}

void oscillation()
// An oscillating wave passing through the cube
{
  uint8_t curZ = 0, dz = 1;

  PRINTS("\nOscillation");

  C.clear();
  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    // shift all the planes back by one
    for (uint8_t x = C.size(MD_Cubo::XAXIS) - 1; x > 0; x--)
      C.copyPlane(MD_Cubo::YZPLANE, x - 1, x);
    C.fillPlane(false, MD_Cubo::YZPLANE, 0);

    // draw the wave line on the YZ plane
    C.drawLine(true, 0, 0, curZ, 0, C.size(MD_Cubo::YAXIS) - 1, C.size(MD_Cubo::ZAXIS) - 1 - curZ);
    curZ += dz;
    if (curZ == C.size(MD_Cubo::ZAXIS) || curZ == 0) dz = -dz;

    C.update();
    C.animate(100);
  }
}

void flagwave()
// Looks like a flag waving in the wind
{
  uint8 t curY = 0, dy = 1;

  PRINTS("\nFlag wave");

  C.clear();
  timeStart = millis();
  while (millis() - timeStart <= DEMO RUNTIME)
  {
    // shift all the planes back by one
    for (uint8_t x = C.size(MD_Cubo::XAXIS) - 1; x > 0; x--)
      C.copyPlane(MD_Cubo::YZPLANE, x - 1, x);
    C.fillPlane(false, MD_Cubo::YZPLANE, 0);

    // draw the wave line on the YZ plane
    C.drawLine(true, 0, curY, 0, 0, curY, C.size(MD_Cubo::ZAXIS) - 1);
    curY += dy;
    if (curY == C.size(MD_Cubo::YAXIS)-1 || curY == 0) dy = -dy;

    C.update();
    C.animate(100);
  }
}

void spiral()
// A spiral wave passing through the cube
{
  uint8_t curZ = 0, dz = 1;
  uint8 t curY = 0, dy = 0;

  PRINTS("\nSpiral");

  C.clear();
  timeStart = millis();
  while (millis() - timeStart <= DEMO RUNTIME)
  {
    // shift all the planes back by one
    for (uint8_t x = C.size(MD_Cubo::XAXIS) - 1; x > 0; x--)
      C.copyPlane(MD_Cubo::YZPLANE, x - 1, x);
```

```
    C.fillPlane(false, MD_Cubo::YZPLANE, 0);

    // draw the spiral line on the YZ plane
    if (dy == 0)
      C.drawLine(true, 0, 0, curZ, 0, C.size(MD_Cubo::YAXIS) - 1, C.size(MD_Cubo::ZAXIS) - 1 - curZ);
    else
      C.drawLine(true, 0, curY, C.size(MD_Cubo::ZAXIS) - 1, 0, C.size(MD_Cubo::YAXIS) - 1 - curY, 0);
    curZ += dz;
    curY += dy;
    // reset the Z - notefirst line of Y and last of Z are the same line so avoid double delays by
stopping
    // Z earlier
    if (curZ == C.size(MD_Cubo::ZAXIS)-1)
    {
      dz = 0;
      dy = 1;
      curZ = 0;
    }
    // reset the Y
    if (curY == C.size(MD_Cubo::YAXIS))
    {
      dz = 1;
      dy = 0;
      curY = 0;
    }

    C.update();
    C.animate(100);
  }
}

void suspension()
// Full plane moving end-to-end losing LEDs 'in suspension' as it goes
{
  const uint16_t delay = 100;
  // Spead out the LEDs evenly in each layer
  const uint8_t numSpeckles = (C.size(MD_Cubo::YAXIS) * C.size(MD_Cubo::ZAXIS)) / C.size(MD_Cubo::XAXIS);

  PRINTS("\nSuspension");

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    C.clear();

    // fill the end plane
    C.fillPlane(true, MD_Cubo::YZPLANE, 0);
    C.update();
    C.animate(delay);

    // * Speaading points through the cube
    // 1. copy the current plane to the next one
    // 2. clear the current plane
    // 3. ramdomly clear numSpeckle points in the new plane and set the same points in the previous one
    for (uint8_t x = 0; x < C.size(MD_Cubo::XAXIS) - 1; x++)
    {
      C.copyPlane(MD_Cubo::YZPLANE, x, x + 1);
      C.fillPlane(false, MD_Cubo::YZPLANE, x);

      for (uint8_t j = 0; j < numSpeckles; j++)
      {
        uint8_t y, z;

        do
        {
          y = random(C.size(MD_Cubo::YAXIS));
          z = random(C.size(MD_Cubo::ZAXIS));
        } while (!C.getVoxel(x+1, y, z));

        C.setVoxel(true, x, y, z);
        C.setVoxel(false, x + 1, y, z);
      }
```

```
      C.update();
      C.animate(delay);
    }

    C.animate(delay * 3);

    // * Collapsing it all together again
    // 1. every point set in this layer needs to be set in the next.
    // 2. clear the current layer.
    for (uint8_t x = 0; x < C.size(MD_Cubo::XAXIS) - 1; x++)
    {
      for (uint8_t y = 0; y < C.size(MD_Cubo::YAXIS); y++)
        for (uint8_t z = 0; z < C.size(MD_Cubo::YAXIS); z++)
          if (C.getVoxel(x, y, z)) C.setVoxel(true, x + 1, y, z);
      C.fillPlane(false, MD_Cubo::YZPLANE, x);
      C.update();
      C.animate(delay);
    }

    C.animate(delay * 3);

    // now do it all again in reverse
    // * Speaading points through the cube
    for (uint8_t x = C.size(MD_Cubo::XAXIS) - 1; x > 0; x--)
    {
      C.copyPlane(MD_Cubo::YZPLANE, x, x - 1);
      C.fillPlane(false, MD_Cubo::YZPLANE, x);

      for (uint8 t j = 0; j < numSpeckles; j++)
      {
        uint8 t y, z;

        do
        {
          y = random(C.size(MD_Cubo::YAXIS));
          z = random(C.size(MD_Cubo::ZAXIS));
        } while (!C.getVoxel(x - 1, y, z));

        C.setVoxel(true, x, y, z);
        C.setVoxel(false, x - 1, y, z);
      }

      C.update();
      C.animate(delay);
    }

    C.animate(delay * 3);

    // * Collapsing it all together again
    for (uint8_t x = C.size(MD_Cubo::XAXIS) - 1; x > 0; x--)
    {
      for (uint8_t y = 0; y < C.size(MD_Cubo::YAXIS); y++)
        for (uint8_t z = 0; z < C.size(MD_Cubo::YAXIS); z++)
          if (C.getVoxel(x, y, z)) C.setVoxel(true, x - 1, y, z);
      C.fillPlane(false, MD_Cubo::YZPLANE, x);
      C.update();
      C.animate(delay);
    }

    C.animate(delay * 3);
  }
}

void hourglass()
// Looks like an hourglass dropping sand grains as time passes
{
#define GP(x, y, z) (((x & 0x7)<<5) | ((y & 0x7)<<2) | (z & 3))
#define UX(b) (b>>5)
#define UY(b) ((b>>2) & 0x7)
#define UZ(b) (b & 0x3)
  static const uint8_t PROGMEM sand[] = // define sand grains in the order they will disappear/appear
  {
    // end layer
```

```
    GP(3, 3, 0), GP(3, 4, 0), GP(4, 4, 0), GP(4, 3, 0), GP(3, 2, 0), GP(2, 3, 0),
    GP(2, 4, 0), GP(3, 5, 0), GP(4, 5, 0), GP(5, 4, 0), GP(5, 3, 0), GP(4, 2, 0),
    GP(3, 1, 0), GP(2, 1, 0), GP(2, 2, 0), GP(1, 1, 0), GP(1, 2, 0), GP(1, 3, 0),
    GP(1, 4, 0), GP(1, 5, 0), GP(2, 5, 0), GP(1, 6, 0), GP(2, 6, 0), GP(3, 6, 0),
    GP(4, 6, 0), GP(5, 6, 0), GP(5, 5, 0), GP(6, 6, 0), GP(6, 5, 0), GP(6, 4, 0),
    GP(6, 3, 0), GP(6, 2, 0), GP(5, 2, 0), GP(6, 1, 0), GP(5, 1, 0), GP(4, 1, 0),

    // end-1 layer
    GP(3, 3, 1), GP(3, 4, 1), GP(4, 4, 1), GP(4, 3, 1), GP(3, 2, 1), GP(2, 3, 1),
    GP(2, 4, 1), GP(3, 5, 1), GP(4, 5, 1), GP(5, 4, 1), GP(5, 3, 1), GP(4, 2, 1),
    GP(3, 1, 1), GP(2, 1, 1), GP(2, 2, 1), GP(1, 1, 1), GP(1, 2, 1), GP(1, 3, 1),
    GP(1, 4, 1), GP(1, 5, 1), GP(2, 5, 1), GP(1, 6, 1), GP(2, 6, 1), GP(3, 6, 1),
    GP(4, 6, 1), GP(5, 6, 1), GP(5, 5, 1), GP(6, 6, 1), GP(6, 5, 1), GP(6, 4, 1),
    GP(6, 3, 1), GP(6, 2, 1), GP(5, 2, 1), GP(6, 1, 1), GP(5, 1, 1), GP(4, 1, 1),

    // end-2 layer
    GP(3, 3, 2), GP(3, 4, 2), GP(4, 4, 2), GP(4, 3, 2), GP(3, 2, 2), GP(2, 2, 2),
    GP(2, 3, 2), GP(2, 4, 2), GP(2, 5, 2), GP(3, 5, 2), GP(4, 5, 2), GP(5, 5, 2),
    GP(5, 4, 2), GP(5, 3, 2), GP(5, 2, 2), GP(4, 2, 2),

    // end-3 layer
    GP(3, 3, 3), GP(3, 4, 3), GP(4, 4, 3), GP(4, 3, 3)
  };
  uint8_t s, x, y, z;
  const uint16_t delay = DEMO_RUNTIME / ARRAY_SIZE(sand) / 3; // 3 delays per action

  // this only works for larger cubes, soreturn if the cube is not big enough
  if ((C.size(MD_Cubo::XAXIS) != 8 || C.size(MD_Cubo::YAXIS) != 8 || C.size(MD_Cubo::ZAXIS) != 8))
    return;

  PRINTS("\nHourglass");

  // Draw the hourglass cube outline and the sand within it
  C.clear();
  C.drawCube(true, 0, 0, 0, C.size(MD_Cubo::XAXIS));
  for (uint8_t i = 0; i < ARRAY_SIZE(sand); i++)
  {
    s = pgm_read_byte(&sand[i]);
    x = UX(s); y = UY(s); z = C.size(MD_Cubo::ZAXIS) - 1 - UZ(s);
    C.setVoxel(true, x, y, z);
  }
  C.update();
  C.animate(1000);

  // drop all the grains to the bottom
  for (uint8_t i = 0; i < ARRAY_SIZE(sand); i++)
  {
    s = pgm_read_byte(&sand[i]);
    x = UX(s); y = UY(s); z = UZ(s);

    // turn it off at the top
    C.setVoxel(false, x, y, C.size(MD_Cubo::ZAXIS) - 1 - z);
    C.update();
    C.animate(delay);

    // drop it down
    for (uint8_t i = (C.size(MD_Cubo::ZAXIS) / 2); i > z; i--)
    {
      C.setVoxel(true, 3, 3, i);
      C.update();
      C.animate(delay / 3);
      C.setVoxel(false, 3, 3, i);
    }

    // turn it on at the bottom
    C.setVoxel(true, x, y, z);

    // update and wait for next grain
    C.update();
    C.animate(delay);
  }
  C.animate(1000);
}
```

```
void boingCube()
// Small cube grows in the direction of one of the zaxes and then expands out to full size,
// repeated in reverse and interates theough all the axes.
{
  const uint16_t  delay = 100;
  uint8_t pass = 0;
  uint8_t x, y, z;  // corner coordinates of the rect prism we are drawing
  int8_t  dx, dy, dz; // direction of expansion for each coordinate
  int8_t  sx, sy, sz; // the size of the cube in x, y and z directions

  PRINTS("\nBoing Cube");

  x = (C.size(MD_Cubo::XAXIS) / 2) - 1;
  y = (C.size(MD_Cubo::YAXIS) / 2) - 1;
  z = (C.size(MD_Cubo::ZAXIS) / 2) - 1;
  sx = 2; sy = 2; sz = 2;

  // draw the initial seed cube
  C.clear();
  C.drawRPrism(true, x, y, z, sx-1, sy-1, sz-1);
  C.update();
  C.animate(delay);

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    // expand the cube along the chosen direction for this path
    switch (pass)
    {
      case 0: dx = -1; dy = 0; dz = 0; break;
      case 1: dx = 0; dy = -1; dz = 0; break;
      case 2: dx = 0; dy = 0; dz = -1; break;
    }

    while (x != 0 && y != 0 && z != 0)  // stop when we have reached the 'front' face
    {
      x += dx; y += dy; z += dz;
      sx += 2 * abs(dx); sy += 2 * abs(dy); sz += 2 * abs(dz);
      C.clear();
      C.drawRPrism(true, x, y, z, sx-1, sy-1, sz-1);
      C.update();
      C.animate(delay);
    }

    // now expand the face at either end until we have a full cube
    switch (pass)
    {
      case 0: dx = 0; dy = -1; dz = -1; break;
      case 1: dx = -1; dy = 0; dz = -1; break;
      case 2: dx = -1; dy = -1; dz = 0; break;
    }

    while (sx != C.size(MD_Cubo::XAXIS) || sy != C.size(MD_Cubo::YAXIS) || sz != C.size(MD_Cubo::ZAXIS))
    {
      x += dx; y += dy; z += dz;
      sx += 2 * abs(dx); if (sx == C.size(MD_Cubo::XAXIS)) dx = 0;
      sy += 2 * abs(dy); if (sy == C.size(MD_Cubo::YAXIS)) dy = 0;
      sz += 2 * abs(dz); if (sz == C.size(MD_Cubo::ZAXIS)) dz = 0;
      C.clear();
      C.drawRPrism(true, x, y, z, sx - 1, sy - 1, sz - 1);
      C.update();
      C.animate(delay);
    }
    C.animate(2*delay);

    // now shrink it back down
    switch (pass)
    {
    case 0: dx = 0; dy = 1; dz = 1; break;
    case 1: dx = 1; dy = 0; dz = 1; break;
    case 2: dx = 1; dy = 1; dz = 0; break;
    }
```

```
    while (sx != 2 && sy != 2 && sz != 2)
    {
      x += dx; y += dy; z += dz;
      sx -= 2 * abs(dx); if (sx == 2) dx = 0;
      sy -= 2 * abs(dy); if (sy == 2) dy = 0;
      sz -= 2 * abs(dz); if (sz == 2) dz = 0;
      C.clear();
      C.drawRPrism(true, x, y, z, sx - 1, sy - 1, sz - 1);
      C.update();
      C.animate(delay);
    }

    switch (pass)
    {
    case 0: dx = 1; dy = 0; dz = 0; break;
    case 1: dx = 0; dy = 1; dz = 0; break;
    case 2: dx = 0; dy = 0; dz = 1; break;
    }

    while (sx != 2 || sy != 2 || sz != 2)  // stop when we have reached the 'front' face
    {
      sx -= 2 * abs(dx); sy -= 2 * abs(dy); sz -= 2 * abs(dz);
      x += dx; if (sx == 2) dx = 0;
      y += dy; if (sy == 2) dy = 0;
      z += dz; if (sz == 2) dz = 0;
      C.clear();
      C.drawRPrism(true, x, y, z, sx - 1, sy - 1, sz - 1);
      C.update();
      C.animate(delay);
    }

    // increment the pass we are on with wraparound
    pass = (pass + 1) % 3;
  }
}

void spiralLine()
// vertical line spirals from the outside to the inside and then moves back out, to start all over again
{
  uint16_t delay = 50;

  PRINTS("\nSpiral Line");

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    for (uint8_t instep = 0; instep < C.size(MD_Cubo::XAXIS) / 2; instep++)
    {
      for (int8_t x = instep; x < C.size(MD_Cubo::XAXIS) - instep; x++)
      {
        C.clear();
        C.drawLine(true, x, instep, 0, x, instep, C.size(MD_Cubo::ZAXIS) - 1);
        C.update();
        C.animate(delay);
      }
      for (int8_t y = 1 + instep; y < C.size(MD_Cubo::YAXIS) - instep; y++)
      {
        C.clear();
        C.drawLine(true, C.size(MD_Cubo::XAXIS) - 1 - instep, y, 0, C.size(MD_Cubo::XAXIS) - 1 - instep,
y, C.size(MD_Cubo::ZAXIS) - 1);
        C.update();
        C.animate(delay);
      }
      for (int8_t x = C.size(MD_Cubo::XAXIS) - 2 - instep; x >= instep; x--)
      {
        C.clear();
        C.drawLine(true, x, C.size(MD_Cubo::YAXIS) - 1 - instep, 0, x, C.size(MD_Cubo::YAXIS) - 1 -
instep, C.size(MD_Cubo::ZAXIS) - 1);
        C.update();
        C.animate(delay);
      }
      for (int8_t y = C.size(MD_Cubo::YAXIS) - 2 - instep; y >= 1 + instep; y--)
```

```
      {
        C.clear();
        C.drawLine(true, instep, y, 0, instep, y, C.size(MD_Cubo::ZAXIS) - 1);
        C.update();
        C.animate(delay);
      }
    }
    for (int8_t i = C.size(MD_Cubo::XAXIS) / 2; i >= 0; i--)
    {
      C.clear();
      C.drawLine(true, i, i, 0, i, i, C.size(MD_Cubo::ZAXIS) - 1);
      C.update();
      C.animate(delay);
    }
  }
}

void outsideStack()
// outside square is drawn up the cube, then collpses into the top, the down the cube, repeats
{
  uint16_t delay = 50;

  PRINTS("\nSpiral Line");

  timeStart = millis();
  while (millis() - timeStart <= DEMO_RUNTIME)
  {
    C.clear();
    for (uint8_t step = 0; step < 2; step++)
    {
      for (int8_t z = 0; z < C.size(MD_Cubo::ZAXIS); z++)
      {
        C.drawRPrism(step == 0, 0, 0, z, C.size(MD_Cubo::XAXIS) - 1, C.size(MD_Cubo::YAXIS) - 1, 1);
        C.update();
        C.animate(delay);
      }
    }
    for (uint8_t step = 0; step < 2; step++)
    {
      for (int8_t z = C.size(MD_Cubo::ZAXIS) - 1; z >= 0; z--)
      {
        C.drawRPrism(step == 0, 0, 0, z, C.size(MD_Cubo::XAXIS) - 1, C.size(MD_Cubo::YAXIS) - 1, 1);
        C.update();
        C.animate(delay);
      }
    }
  }
}

void displayChar(char c, uint16_t delay)
// Display a character in the cube
{
  uint8_t size;
  uint8_t cBuf[C.size(MD_Cubo::YAXIS)];  // column data for characters

  C.clear();

  // Get the character defined from the font table and place it in the
  // first YZ plane (x=0)
  size = getFontChar(c, ARRAY_SIZE(cBuf), cBuf);

  // Starting with a clean buffer, only need to set the bits that are on and
  // center the text by adding an offset to the Z coordinate
  for (uint8_t y = 0; y < size; y++)
    for (uint8_t z = 0; z < 8; z++)    // only 8 bits in the height of the font
      if (cBuf[y] & (1 << z))
        C.setVoxel(true, 0, y + (C.size(MD_Cubo::YAXIS) - size) / 2, C.size(MD_Cubo::ZAXIS) - 1 - z);

  // Move the character through the X planes front to back. As an added
  // effect also vary the intensity from max to min.
  for (uint8_t i = 0; i < C.size(MD_Cubo::XAXIS); i++)
  {
    // display the update
```

```
    C.update();
    C.setIntensity(MAX_INTENSITY - (((MAX_INTENSITY / C.size(MD_Cubo::XAXIS)) * i)));
    C.animate(delay);

    // copy to next plane and delete original
    C.copyPlane(MD_Cubo::YZPLANE, i, i + 1);
    C.fillPlane(false, MD_Cubo::YZPLANE, i);
  }
}

void recedingText()
{
  const char  message[] = "MDCuboDemo";
  char  *cp = (char *)message;

  // this only works for larger cubes, soreturn if the cube is not big enough
  if ((C.size(MD_Cubo::XAXIS) < 8 || C.size(MD_Cubo::YAXIS) < 8 || C.size(MD_Cubo::ZAXIS) < 8))
    return;

  PRINTS("\nReceding text");

  while (*cp != '\0')
    displayChar(*cp++, 100);

  C.setIntensity(MAX INTENSITY / 2);
}

void scrollFaces(void)
// scroll all the columns across, starting with the second last and working backwards
{
  // XZ Plane at Ymax
  for (int8_t x = 1; x < C.size(MD_Cubo::XAXIS); x++)
    for (uint8_t z = 0; z < C.size(MD_Cubo::ZAXIS); z++)
    {
      boolean p = C.getVoxel(x + 1, C.size(MD_Cubo::YAXIS) - 1, z);
      C.setVoxel(p, x, C.size(MD_Cubo::YAXIS) - 1, z);
    }
  // YZ Plane at Xmax
  for (int8_t y = C.size(MD_Cubo::YAXIS) - 2; y >= 0; y--)
    for (uint8_t z = 0; z < C.size(MD_Cubo::ZAXIS); z++)
    {
      boolean p = C.getVoxel(C.size(MD_Cubo::XAXIS) - 1, y, z);
      C.setVoxel(p, C.size(MD_Cubo::XAXIS) - 1, y + 1, z);
    }
  // XZ Plane at Y0
  for (int8_t x = C.size(MD_Cubo::XAXIS) - 2; x >= 0; x--)
    for (uint8_t z = 0; z < C.size(MD_Cubo::ZAXIS); z++)
    {
      boolean p = C.getVoxel(x, 0, z);
      C.setVoxel(p, x + 1, 0, z);
    }
  // YZ Plane at X0
  for (int8_t y = 1; y < C.size(MD_Cubo::YAXIS); y++)
    for (uint8_t z = 0; z < C.size(MD_Cubo::ZAXIS); z++)
    {
      boolean p = C.getVoxel(0, y, z);
      C.setVoxel(p, 0, y - 1, z);
    }
}

// States for FSM
#define S_INIT  0
#define S_LOADC 1
#define S_SCROLL  2
#define S_WAITING 3
#define S_TIMER   4
#define S_CLEANUP 5
#define S_END   6

boolean displayMessage(char *mesg, uint16_t delay = 0)
// Display a message on the surface of the cube, scrolling around the outside faces
// Implemented as a non-blocking Finte State Machine.
// Returns true when the message has completed.
```

```
// The message is only reference by a pointer to the string,
// so the caller is responsible for making sure it remains stable
// until the data is displayed.
{
  static uint8_t state = S_END;
  static uint32_t timeStart;    // time at the start of the wait (ms)
  static uint16_t timeWait;     // waiting time (ms)
  static char  *cp;             // where we are in the message
  static uint8 t cBuf[10];      // column data for characters - this should be bigger than any of the
faces being displayed
  static uint8_t getCol = 0;    // current column of the character
  static uint8_t size;          // size of the character (columns)
  static uint16 t countScroll;  // the number of columns being scrolled

  if (mesg != NULL) state = S_INIT; // new message to process

  switch (state)
  {
  case S_INIT: // initialisation of a new message
    PRINTS("\nS_INIT");
    cp = mesg;
    timeWait = delay;
    countScroll = (C.size(MD_Cubo::XAXIS) - 1 + C.size(MD_Cubo::YAXIS) - 1) * 2;
    state = S_LOADC;
    break;

  case S_LOADC: // load a character from the font table
    PRINTS("\nS_LOADC");
    if (*cp == '\0')  // reached end of message
    {
      PRINTS(" - end mesg reached");
      state = S_CLEANUP;
    }
    else                // load a character into the buffer
    {
      PRINT(" - loading '", *cp);
      size = getFontChar(*cp++, ARRAY_SIZE(cBuf), cBuf);
      cBuf[size++] = 0;   // inter character blank column
      PRINT("' size ", size);
      getCol = 0;
      state = S_SCROLL;
    }
    break;

  case S_SCROLL: // insert a column into the start and scroll along
    PRINTS("\nS_SCROLL");
    scrollFaces();

    // add in the new column
    for (uint8_t z = 0; z < 8; z++)    // only 8 bits in the height of the font
      C.setVoxel((cBuf[getCol] & (1 << z)) != 0, 0, C.size(MD_Cubo::YAXIS) - 1, C.size(MD_Cubo::ZAXIS) - 1
- z);
    getCol++;

    // update the display
    C.update();
    C.animate(0);
    state = S_WAITING;
    break;

  case S_WAITING: // initialisation stage for S_TIMER
    timeStart = millis();
    state = S_TIMER;
    break;

  case S_TIMER: // non blocking wait for the scrolling time to expire
    //PRINTS("\nS_WAITING");
    // finished waiting?
    if (millis() - timeStart < timeWait)
    {
      C.animate(0);
      break;
    }
```

```
    // move to a state to do something
    PRINTS("\nS WAITING Over");
    if (getCol == size) // all the columns processed
    {
      if (*cp == '\0')  // no characters left - clean up
        state = S_CLEANUP;
      else              // otherise get the next char
        state = S LOADC;
    }
    else                // more columns to process - do it
      state = S_SCROLL;
    break;

  case S_CLEANUP: // keep scrolling until the message disappears
    PRINTS("\nS_CLEANUP");
    scrollFaces();
    C.update();
    C.animate(0);

    if (countScroll-- != 0)
      state = S_WAITING;
    else
      state = S_END;
    break;

  case S_END:   // nothing left to do except wait for another message
    // PRINTS("\nS_END");
    break;
  }

  return (state == S_END);
}

void scrollingText()
{
  char message[] = "MD Cubo          Demo";

  // this only works for larger cubes, soreturn if the cube is not big enough
  if ((C.size(MD_Cubo::XAXIS) < 8 || C.size(MD_Cubo::YAXIS) < 8 || C.size(MD_Cubo::ZAXIS) < 8))
    return;

  PRINTS("\nScrolling Text")
  C.clear();    // new clean cube
  displayMessage(message, 100);

  while (!displayMessage(NULL))
    ;
}

void setup()
{
#if DEBUG
  Serial.begin(57600);
  PRINTS("\n[MD_Cubo Demo]");
#endif
  C.begin();

  randomSeed(analogRead(A0) * analogRead(A5));  // sort of gives different start results.
}

void loop()
{
  static int8_t old_choice = -1;

  void (*demoType[])(void) =
  {
    oscillation, firefly, intersectPlanes, brownian,
    ripples, outsideStack, recedingText, rain, shrinkCube, randomFill, flagwave,
    slideFaces, hourglass, scrollingText, wrapFaces,
    suspension, spiralLine, boingCube, droplets
  };
```

```
#if RANDOM_CYCLE
  uint8_t choice;

  do
    choice = random(ARRAY_SIZE(demoType));
  while (choice == old_choice);
  old_choice = choice;
#else
  old_choice = (old_choice + 1) % ARRAY_SIZE(demoType);
#endif // RANDOM_CYCLE

  demoType[old_choice]();
}
```